



华章科技

资深Hadoop技术专家撰写，EasyHadoop和51CTO等专业技术社区联袂推荐！

从源代码角度深入分析MapReduce的设计理念，以及RPC框架、客户端、JobTracker、TaskTracker和Task等运行时环境的架构设计与实现原理。

深入探讨Hadoop性能优化、多用户作业调度器、安全机制、下一代MapReduce框架等高级主题。



技术丛书



Hadoop Internals: in-depth study of MapReduce

Hadoop技术内幕

深入解析MapReduce架构设计与实现原理

董西成◎著



机械工业出版社
China Machine Press

目 录

前言

为什么要写这本书

读者对象

如何阅读本书

勘误和支持

致谢

第一部分 基础篇

第1章 阅读源代码前的准备

1.1 准备源代码学习环境

1.2 获取Hadoop源代码

1.3 搭建Hadoop源代码阅读环境

1.4 Hadoop源代码组织结构

1.5 Hadoop初体验

1.6 编译及调试Hadoop源代码

1.7 小结

第2章 MapReduce设计理念与基本架构

2.1 Hadoop发展史

2.2 Hadoop MapReduce设计目标

2.3 MapReduce编程模型概述

2.4 Hadoop基本架构

2.5 Hadoop MapReduce作业的生命周期

2.6 小结

第二部分 MapReduce编程模型篇

第3章 MapReduce编程模型

3.1 MapReduce编程模型概述

3.2 MapReduce API基本概念

3.3 Java API解析

3.4 非Java API解析

3.5 Hadoop工作流

3.6 小结

第三部分 MapReduce核心设计篇

第4章 Hadoop RPC框架解析

4.1 Hadoop RPC框架概述

4.2 Java基础知识

4.3 Hadoop RPC基本框架分析

4.4 MapReduce通信协议分析

4.5 小结

第5章 作业提交与初始化过程分析

5.1 作业提交与初始化概述

- 5.2 作业提交过程详解
- 5.3 作业初始化过程详解
- 5.4 Hadoop DistributedCache原理分析
- 5.5 小结

第6章 JobTracker内部实现剖析

- 6.1 JobTracker概述
- 6.2 JobTracker启动过程分析
- 6.3 心跳接收与应答
- 6.4 Job和Task运行时信息维护
- 6.5 容错机制
- 6.6 任务推测执行原理
- 6.7 Hadoop资源管理
- 6.8 小结

第7章 TaskTracker内部实现剖析

- 7.1 TaskTracker概述
- 7.2 TaskTracker启动过程分析
- 7.3 心跳机制
- 7.4 TaskTracker行为分析
- 7.5 作业目录管理
- 7.6 启动新任务
- 7.7 小结

第8章 Task运行过程分析

- 8.1 Task运行过程概述
- 8.2 基本数据结构和算法
- 8.3 Map Task内部实现
- 8.4 Reduce Task内部实现
- 8.5 Map/Reduce Task优化
- 8.6 小结

第四部分 MapReduce高级篇

第9章 Hadoop性能调优

- 9.1 概述
- 9.2 从管理员角度进行调优
- 9.3 从用户角度进行调优
- 9.4 小结

第10章 Hadoop多用户作业调度器

- 10.1 多用户调度器产生背景
- 10.2 HOD
- 10.3 Hadoop队列管理机制
- 10.4 Capacity Scheduler实现
- 10.5 Fair Scheduler实现

10.6 其他Hadoop调度器介绍

10.7 小结

第11章 Hadoop安全机制

11.1 Hadoop安全机制概述

11.2 基础知识

11.3 Hadoop安全机制实现

11.4 应用场景总结

11.5 小结

第12章 下一代MapReduce框架

12.1 第一代MapReduce框架的局限性

12.2 下一代MapReduce框架概述

12.3 Apache YARN

12.4 Facebook Corona

12.5 Apache Mesos

12.6 小结

附录A 安装Hadoop过程中可能存在的问题及解决方案

附录B Hadoop默认HTTP端口号以及HTTP地址

参考资料

前言

为什么要写这本书

突然之间，大数据一下子就“火”了，开源软件Hadoop也因此水涨船高。得益于一些国际领先厂商，尤其是FaceBook、Yahoo! 以及阿里巴巴等互联网巨头的现身说法，Hadoop被看成大数据分析的“神器”。IDC在对未来几年的预测中就专门提到了大数据，其认为未来几年，会有越来越多的企业级用户试水大数据平台和应用，而这之中，Hadoop将成为最耀眼的“明星”。

尽管Hadoop整个生态系统是开源的，但是，由于它包含的软件种类过多，且版本升级过快，大部分公司，尤其是一些中小型企业公司，难以在有限的时间内快速掌握Hadoop蕴含的价值。此外，Hadoop自身版本的多样化也给很多研发人员带来了很大的学习负担。尽管当前市面上已有很多参考书籍，比如《Hadoop: The Definitive Guide》、《Hadoop in Action》、《Pro Hadoop》、《Hadoop Operations》等，但是，至今还没有一本书能够深入地剖析Hadoop内部的实现细节，比如JobTracker实现、作业调度器实现等。也正因如此，很多Hadoop初学者和研发人员只能参考网络上一些零星的源代码分析的文章，自己一点一点地阅读源代码，缓慢地学习Hadoop。而本书正是为了解决以上各种问题而编写的，它是国内第一本深入剖析Hadoop内部实现细节的书籍。

本书以Hadoop 1.0为基础，深入剖析了Hadoop MapReduce中各个组件的实现细节，包括RPC框架、JobTracker实现、TaskTracker实现、Task实现和作业调度器实现等。书中不仅详细介绍了MapReduce各个组件的内部实现原理，而且结合源代码进行了深入的剖析，使读者可以快速全面地掌握Hadoop MapReduce设计原理和实现细节。

读者对象

（1）Hadoop二次开发人员

Hadoop由于在扩展性、容错性和稳定性等方面的诸多优点，已被越来越多的公司采用。而为了减少开发成本，大部分公司在Hadoop基础上进行了二次开发，以打造属于公司内部Hadoop平台。对于Hadoop二次开发人员来说，深入而又全面地了解Hadoop的设计原理与实现细节是修改Hadoop内核的前提，而本书可帮助这部分读者快速而又全面地了解Hadoop实现细节。

（2）Hadoop应用开发人员

如果要利用Hadoop进行高级应用开发，仅掌握Hadoop基本使用方法是远远不够的，必须对Hadoop框架的设计原理、架构和运作机制有一定的了解。对这部分读者而言，本书将带领他们全面了解Hadoop的设计和实现原理，加深对Hadoop框架的理解，提高开发水平，从而编写出更加高效的MapReduce应用程序。

（3）Hadoop运维工程师

对于一名合格的Hadoop运维工程师而言，适当地了解Hadoop框架的设计原理、架构和运作机制是十分有帮助的。这不仅可以使Hadoop运维人员更快地排除各种可能的Hadoop故障，还可以让Hadoop运维人员与研发人员进行更有效的沟通。通过阅读这本书，Hadoop运维人员可以了解到很多其他书中无法获取的Hadoop实现细节。

（4）开源软件爱好者

Hadoop是开源软件中的佼佼者。它在实现的过程中吸收了很多开源领域的优秀思想，同时有很多值得学习的创新。尤为值得一提的是，本书分析Hadoop架构设计和实现原理的方式也许值得所有开源软件爱好者学习和借鉴。通过阅读本书，这部分读者不仅能领略到开源软件的优秀思想，还可以掌握分析开源软件源代码的方法和技巧，从而进一步提高使用开源软件的效率和质量。

如何阅读本书

本书分为四大部分（不包括附录）：

第一部分为基础篇，简单地介绍Hadoop的阅读环境搭建和基本设计架构，帮助读者了解一些基础背景知识。

第二部分为MapReduce编程模型篇，着重讲解MapReduce编程接口，主要包括两套编程接口，分别是旧API和新API。

第三部分为MapReduce核心设计篇，主要讲解Hadoop MapReduce的运行环境，包括RPC框架、客户端、JobTracker、TaskTracker和Task等内部实现细节。

第四部分为MapReduce高级篇，主要讲解Hadoop MapReduce中的一些高级特性和未来发展趋势，包括多用户作业调度器、安全机制和下一代MapReduce框架等。

另外，本书最后还添加了几个附录：附录A为安装Hadoop过程中可能存在的问题及解决方案；附录B为Hadoop默认HTTP端口号以及HTTP地址。参考资料中包括了本书写作过程中参考的书籍、论文、Hadoop Jira和网络资源。

如果你是一名经验丰富的资深用户，能够理解Hadoop的相关基础知识和使用技巧，那么你可以直接阅读第三部分和第四部分。但是，如果你是一名初学者，请一定从第1章的基础理论知识开始学习。

勘误和支持

由于笔者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，笔者特意创建了一个在线支持与应急方案的站点<http://hadoop123.com>。你可以将书中的错误发布在Bug勘误表页面中。如果你遇到问题，可以访问Q&A页面，我将尽量在线上为读者提供最满意的解答。如果你有什么宝贵意见，欢迎发送邮件至dongxicheng@yahoo.com，期待能够得到你的真挚反馈。

致谢

感谢我的导师廖华明副研究员。在我读研没空顾及项目的时候，她给了我一次又一次的鼓励，她甚至专门为我写书留出空闲时间。在廖老师的身边，我还学会了很多专业知识以外的东西。

感谢腾讯的蔡斌老师。正是由于他的推荐，才使本书的出版成为可能。

感谢机械工业出版社华章公司的杨福川老师和孙海亮老师。是他们在这一年的时间中始终支持着我的写作，是他们的鼓励和帮助使我顺利完成了本书的编写工作。

感谢对本书部分章节提出改进建议的何鹏、姜冰、郑伟伟等人。另外，感谢给我提供各种帮助的战科宇、周礼、刘晏辰、孟椿智、王群、王颖、曹聪、朱雪峰等人。

最后，感谢我父母的养育之恩，感谢兄长的鼓励和支持，感谢他们时时刻刻给我信心和力量！感谢我的女朋友颢悦对我生活的悉心照料与琐事上的宽容。

谨以此书献给我最亲爱的家人，以及众多热爱Hadoop的朋友们！

董西成

于北京

第一部分 基础篇

本部分内容

阅读源代码前的准备

MapReduce设计理念与基本架构

第1章 阅读源代码前的准备

一般而言，在深入研究一个系统的技术细节之前，先要进行一些基本的准备工作，比如，准备源代码阅读环境，搭建运行环境并尝试使用该系统等。对于Hadoop而言，由于它是一个分布式系统，且由多种守护进程组成，具有一定的复杂性，如果想深入学习其设计原理，仅仅进行以上几项准备工作是不够的，还要学习一些调试工具的使用方法，以便对Hadoop源代码进行调试、跟踪。边用边学，这样才能事半功倍。

本章的编写目的是帮助读者构建一个“高效”的Hadoop源代码学习环境，包括Hadoop源代码阅读环境、Hadoop使用环境和Hadoop源代码编译调试环境等，这主要涉及如下内容：

- 在Linux和Windows环境下搭建Hadoop源代码阅读环境的方法；
- Hadoop的基本使用方法，主要包括Hadoop Shell和Eclipse插件两种工具的使用；
- Hadoop源代码编译和调试方法，其中，调试方法包括使用Eclipse远程调试和打印调试日志两种。

考虑到大部分用户在单机上学习Hadoop源代码，所以本章内容均是基于单机环境的。本章大部分内容较为基础，已经掌握这部分内容的读者可以直接跳过本章。

1.1 准备源代码学习环境

对于大部分公司而言，实验和生产环境中的服务器集群部署的都是Linux操作系统。考虑到Linux在服务器市场中具有统治地位，Hadoop从一开始便是基于Linux操作系统开发的，因而对Linux有着非常完美的支持。尽管Hadoop采用了具有跨平台特性的Java作为主要编程语言，但由于它的一些功能实现用到了Linux操作系统相关的技术，因而对其他平台的支持不够友好，且没有进行过严格测试。换句话说，其他操作系统（如Windows）仅可作为开发环境^[1]，不可作为生产环境。对于学习源代码而言，操作系统的选择显得不是非常重要，读者可根据个人爱好自行决定。本节以64 bit Linux和32 bit Windows两种操作系统为例，介绍如何在单机上准备Hadoop源代码学习环境。

1.1.1 基础软件下载

前面提到Hadoop采用的开发语言主要是Java，因而搭建Hadoop环境所需的最基础的软件首先应该包括Java基础开发包JDK和Java编译工具Ant。考虑到源代码阅读和调试的便利性，本书采用功能强大的集成开发环境Eclipse。此外，如果读者选择Windows平台搭建学习环境，还需要安装Cygwin以模拟Linux环境，这是因为Hadoop采用Bash Shell脚本管理集群。搭建Hadoop阅读环境需要的各种软件以及下载方式如表1-1所示。

表 1-1 搭建 Hadoop 阅读环境所需的各种软件及下载方式

软 件	下载网址	推荐版本	说 明
JDK	http://www.oracle.com/technetwork/java/javase/downloads/index.html	1.6 以上	Windows 和 Linux 的安装包不同
Ant	http://ant.apache.org/bindownload.cgi	1.6.0 以上	Windows 和 Linux 使用相同的安装包
Cygwin	http://www.cygwin.com/	最新版本	只有 Windows 平台需要
Eclipse	http://www.eclipse.org/downloads/	Galileo 或者 Helios 版本 [⊖]	Windows 和 Linux 的安装包不同

[1] 截至本书结稿时，Apache Hadoop SVN中已经出现了针对Windows操作系统的分支，具体见<http://svn.apache.org/repos/asf/hadoop/common/branches/>下的branch-1-win和branch-trunk-win，且Hortonworks公司发布了Windows安装版本，具体见<http://hortonworks.com/partners/microsoft/>

[2] 注意，Indigo及以上版本与Hadoop Eclipse插件可能存在兼容问题。

1.1.2 如何准备Windows环境

本小节将介绍如何准备Windows下的Hadoop学习环境，包括JDK、Ant、Cygwin和Eclipse等基础软件的使用方法。本小节假设用户的软件安装目录为D:\hadoop，且最终安装完成的目录结构为：

```
D:\hadoop
├── apache-ant-1.7.1
├── cygwin
└── Java
    └── jdk1.6.0_25
```

1.JDK的安装

用户下载的安装包为jdk-6u25-windows-i586.exe，直接双击该安装包将JDK安装到D:\hadoop\Java\下，然后设配置环境变量JAVA_HOME、CLASSPATH、PATH（不区分大小写），方法如下。

(1) 配置JAVA_HOME

如图1-1所示，在Windows桌面上，右击“计算机”图标，然后在弹出的快捷菜单中依次选择“属性”→“高级系统设置”→“环境变量”命令，然后在系统变量栏，单击“新建”按钮，在弹出的对话框中的“变量名”文本框中填写JAVA_HOME，在“变量值”文本框中填写D:\hadoop\Java\jdk1.6.0_25，然后单击“确定”按钮。



图 1-1 Windows环境下设置JAVA_HOME环境变量

(2) 配置CLASSPATH

参考JAVA_HOME另建一个系统变量，变量名为CLASSPATH，变量值为：

```
.; %JAVA_HOME%/lib/dt.jar; %JAVA_HOME%/lib/tools.jar;
```

(3) 配置PATH

PATH变量已经存在，选中后再单击“编辑”按钮即可。在变量值中添加如下内容：

```
%JAVA_HOME%\bin; %JAVA_HOME%\jre\bin
```

经过以上配置，JDK已经安装完毕，可在DOS窗口中输入命令“java-version”以验证是否安装成功。如果输出以下内容，则说明安装成功：

```
java version"1.6.0_25"  
Java (TM) SE Runtime Environment (build 1.6.0_25-b05)  
Java HotSpot (TM) Client VM (build 20.6-b01, mixed mode, sharing)
```

2.Ant的安装

假设下载的安装包为apache-ant-1.7.1-bin.zip，直接将其解压到工作目录D:\hadoop\下，并添加新的环境变量ANT_HOME，设置其值为D:\hadoop\apache-ant-1.7.1，同时在环境变量PATH后面添加如下内容：

```
; %ANT_HOME%\bin
```

经过以上配置，Ant已经安装完毕，可在DOS窗口中输入命令“ant-version”以验证是否安装成功。如果输出以下内容，则说明安装成功：

```
Apache Ant version 1.7.1 compiled on June 27 2008
```

3.Cygwin的安装

(1) 安装Cygwin

双击下载的Cygwin的安装包setup.exe，一直单击“下一步”按钮，直到出现如图1-2所示的界面，在“Net”一栏中选中OpenSSH相关软件包，会出现如图1-3所示的界面，然后单击“下一步”按钮，此时系统开始在线下载并安装Cygwin环境（时间比较长）。

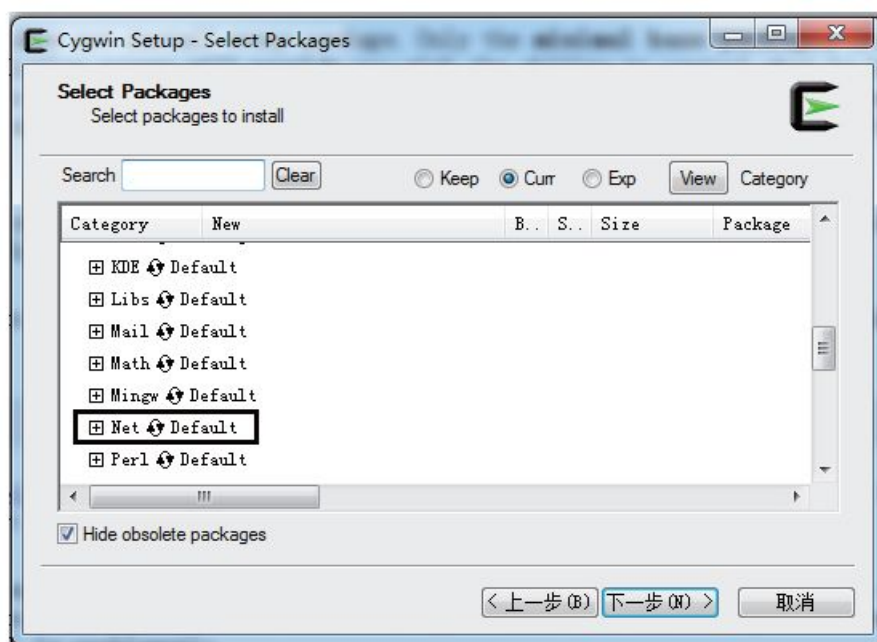


图 1-2 Windows环境下通过Cygwin安装OpenSSH——单击“Net”一栏

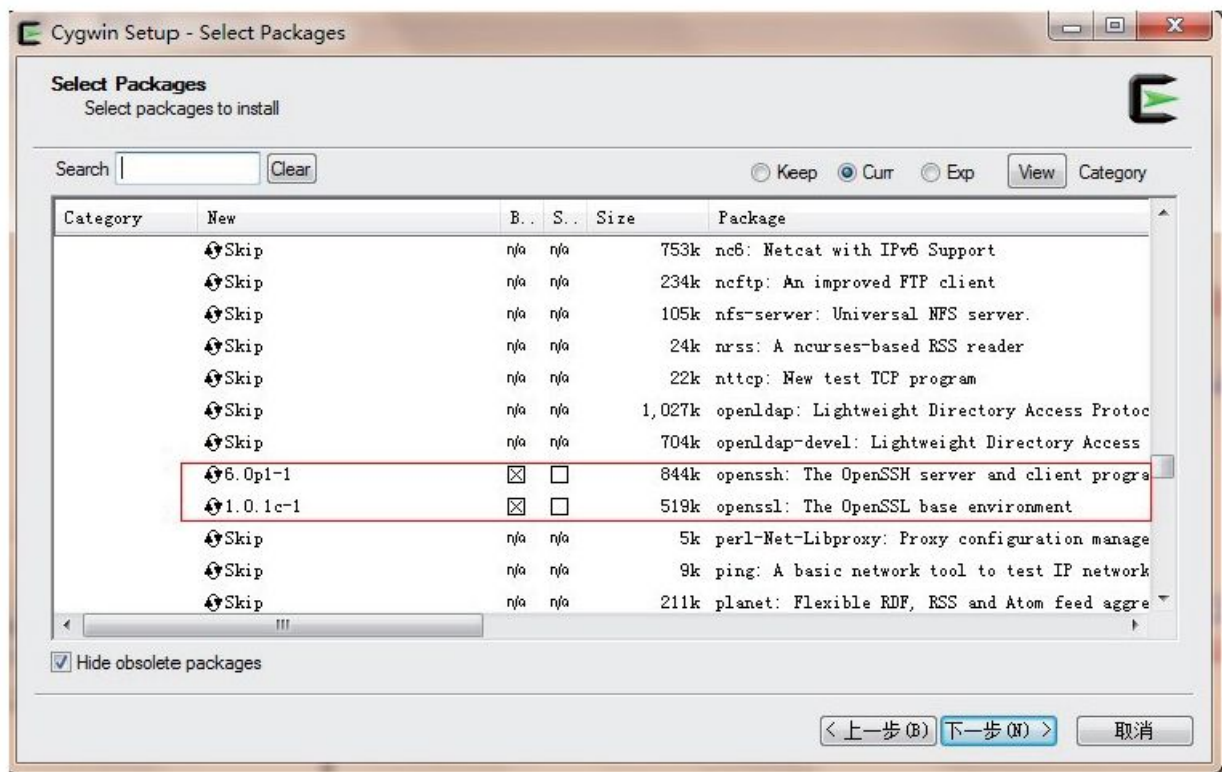


图 1-3 Windows环境下通过Cygwin安装OpenSSH——选中OpenSSH软件包

(2) 安装并启动sshd服务

Hadoop启动/停止脚本需要通过SSH发送命令启动相关守护进程，为此需要安装sshd服务。安装sshd服务的方法是，以管理员身份打开Cygwin命令行终端（右击运行图标，单击“以管理员身份运行”命令），然后输入以下命令：

```
ssh-host-config
```

接着，按照命令行中的提示进行安装，具体如图1-4所示。

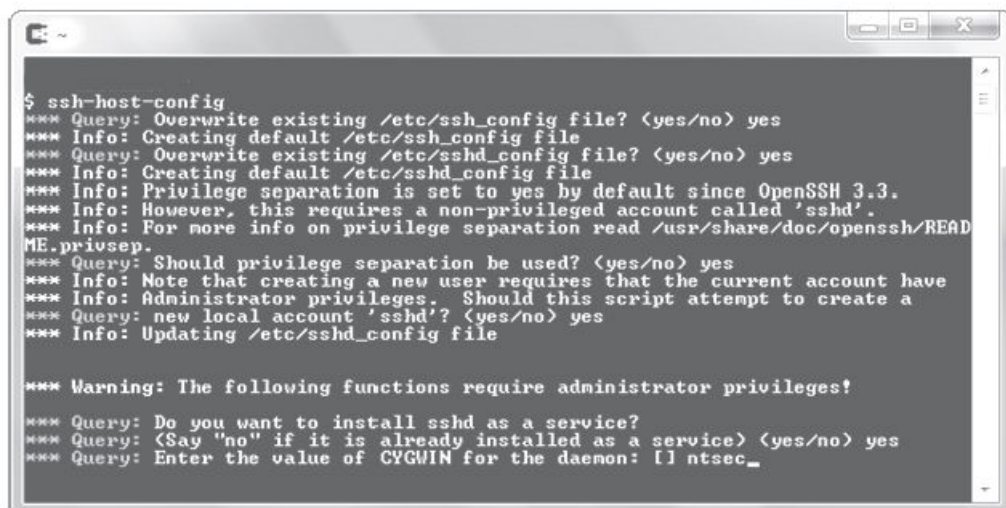


图 1-4 Windows环境下通过Cygwin安装sshd服务

安装完毕后，输入以下命令启动sshd服务：

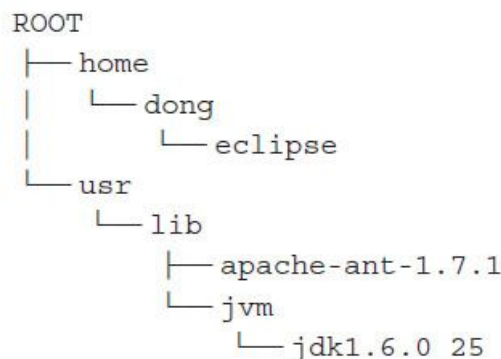
```
net start sshd
```

4.Eclipse的安装

Eclipse官网提供的Eclipse版本均是免安装版，直接将下载的压缩包解压到“D:\hadoop\”下即可使用。

1.1.3 如何准备Linux环境

本小节将介绍如何准备Linux下的Hadoop学习环境。搭建Linux学习环境需要安装JDK, Ant和Eclipse等软件。本书以64 bit Ubuntu为例，介绍安装这些软件的方法，最终安装完成的目录结构为：



1.JDK的安装与配置

一般而言，Ubuntu系统会自带JDK，如果没有或者版本不符合要求，可按以下步骤进行安装：

步骤1 安装JDK。将下载的.bin文件复制到Linux的某个目录下，比如usr/lib/jvm/下，然后在Shell中执行以下命令为该文件添加可执行权限：

```
chmod+x/usr/lib/jvm/jdk1.6.0_25.bin
```

然后执行以下命令安装JDK：

```
sudo/usr/lib/jvm/jdk1.6.0_25.bin
```

之后将会出现安装信息，直至屏幕显示要求按下回车键。此时按下回车键后，会把JDK解压到文件夹jdk1.6.0_25中。至此，JDK已安装完毕，下面进行配置。

步骤2 配置JDK。修改/etc/profile文件，在里面添加以下内容：

```
export JAVA_HOME=/usr/lib/jvm/jdk1.6.0_25
export PATH=$PATH: $JAVA_HOME/bin
export CLASSPATH=$CLASSPATH: $JAVA_HOME/lib: $JAVA_HOME/jre/lib
```

输入以下命令让配置生效：

```
source/etc/profile
```

步骤3 修改默认JDK版本。Ubuntu中可能会有默认的JDK，如openjdk，因而我们需要将自己安装的JDK设置为默认JDK版本，执行下面代码：

```
sudo update-alternatives--install/usr/bin/java java/usr/lib/jvm/jdk1.6.0_25/bin/java 300
sudo update-alternatives--install/usr/bin/javac javac/usr/lib/jvm/jdk1.6.0_25/bin/javac 300
sudo update-alternatives--install/usr/bin/jar jar/usr/lib/jvm/jdk1.6.0_25/bin/jar 300
sudo update-alternatives--install/usr/bin/javah javah/usr/lib/jvm/jdk1.6.0_25/bin/javah 300
sudo update-alternatives--install/usr/bin/javap javap/usr/lib/jvm/jdk1.6.0_25/bin/javap 300
```

然后执行以下代码选择我们安装的JDK版本：

```
sudo update-alternatives--config java
```

步骤4 验证JDK是否安装成功。重启Shell终端，执行命令“java-version”。如果输出以下内容，则说明安装成功：

```
java version"1.6.0_25"  
Java (TM) SE Runtime Environment (build 1.6.0_25-b06)  
Java HotSpot (TM) Client VM (build 20.0-b11, mixed mode, sharing)
```

2.Ant以及Eclipse的安装

（1）安装与配置Ant

首先解压下载包，比如解压到文件/usr/lib/apache-ant-1.7.1目录下，然后修改/etc/profile文件，在里面添加以下内容：

```
export ANT_HOME=/usr/lib/apache-ant-1.7.1  
export PATH=$PATH$: $ANT_HOME/bin
```

输入以下命令让配置生效：

```
source/etc/profile
```

同Windows下的验证方式一样，重启终端，执行命令“ant-version”。如果输出以下内容，则说明安装成功：

```
Apache Ant version 1.7.1 compiled on June 27 2008
```

（2）安装Eclipse

同Windows环境下的安装方式一样，直接解压即可使用。

1.2 获取Hadoop源代码

当前比较流行的Hadoop源代码版本有两个：Apache Hadoop和Cloudera Distributed Hadoop（CDH）。Apache Hadoop是由Yahoo!、Cloudera、Facebook等公司组成的Hadoop社区共同研发的，它属于最原始的开源版本。在该版本的基础上，很多公司进行了封装和优化，推出了自己的开源版本，其中，最有名的一个是Cloudera公司发布的CDH版本。

考虑到Apache Hadoop是最原始的版本，且使用最为广泛，因而本书选用了Apache Hadoop版本为分析对象。自从Apache Hadoop发布以来，已经陆续推出很多版本（具体介绍见2.1.3节）。其中，最具有标志性的版本是1.0.0，而该书正是基于该版本对Hadoop MapReduce进行深入分析的。Hadoop 1.0.0可从<http://hadoop.apache.org/common/releases.html>或<http://svn.apache.org/repos/asf/hadoop/common/branches/branch-1.01>处下载。

1.3 搭建Hadoop源代码阅读环境

1.3.1 创建Hadoop工程

本小节介绍如何创建一个Hadoop源代码工程，以方便阅读源代码。创建一个Hadoop工程，可分两个步骤完成：

步骤1 解压Hadoop源代码。将下载到的Hadoop源代码压缩包hadoop-1.0.0.tar.gz解压到工作目录下（对于Windows系统而言，为了操作方便，解压到Cygwin安装目录的home/{USER}文件夹下）。

步骤2 新建Java工程。打开Eclipse，进入Eclipse可视化界面后，如图1-5所示，依次单击“File”→“New”→“Java Project”，并在弹出的对话框中取消选中“Use default location”前的勾号，然后选择Hadoop安装目录的位置。默认情况下，工程名称与Hadoop安装目录名称相同，用户可自行修改。单击完成按钮，Hadoop源代码工程创建完毕。

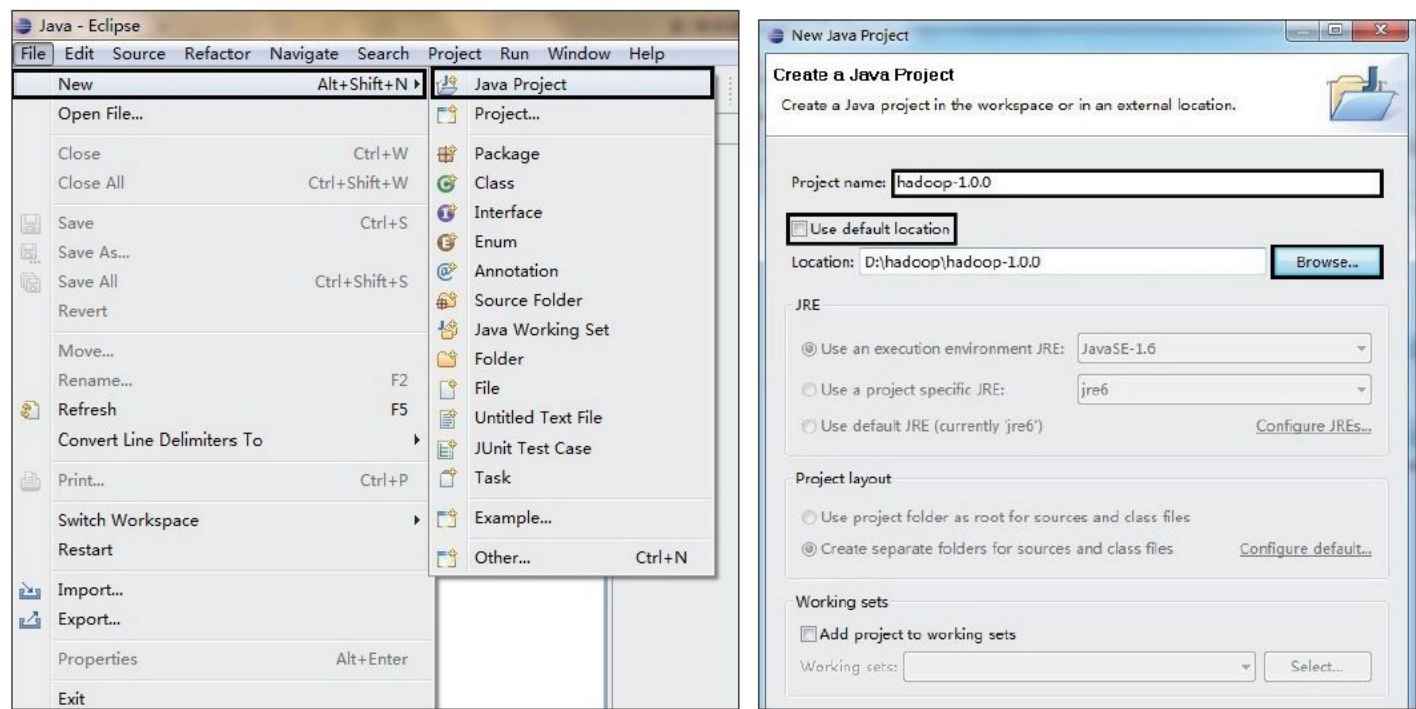


图 1-5 新建Hadoop工程

回到Eclipse主界面后，打开新建的Hadoop工程，可看到整个工程的组织代码，如图1-6所示，源代码按目录组织，且每个目录下以jar包为单位显示各个Java文件。

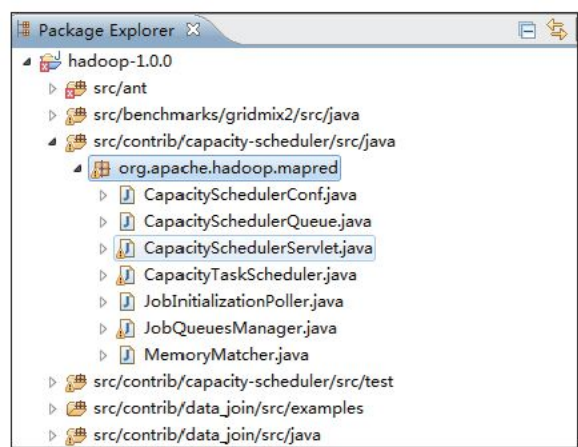


图 1-6 Hadoop工程展示（部分）源代码方式

除了使用源代码压缩包导入Eclipse工程的方法外，读者可以尝试直接从Hadoop SVN上导入Hadoop源代码。这些源代码本身已经是Eclipse工程导出的，Hadoop SVN地址为：<http://svn.apache.org/repos/asf/hadoop/common/branches/>。

1.3.2 Hadoop源代码阅读技巧

本小节介绍在Eclipse下阅读Hadoop源代码的一些技巧，比如如何查看一个基类有哪些派生类、一个方法被其他哪些方法调用等。

1.查看一个基类或接口的派生类或实现类

在Eclipse中，选中某个基类或接口名称，右击，在弹出的快捷菜单中选择“Quick Type Hierarchy”，可在新窗口中看到对应的所有派生类或实现类。

例如，如图1-7所示，打开src\mapred目录下org.apache.hadoop.mapred包中的InputFormat.java文件，查看接口InputFormat的所有实现类，结果如图1-8所示。

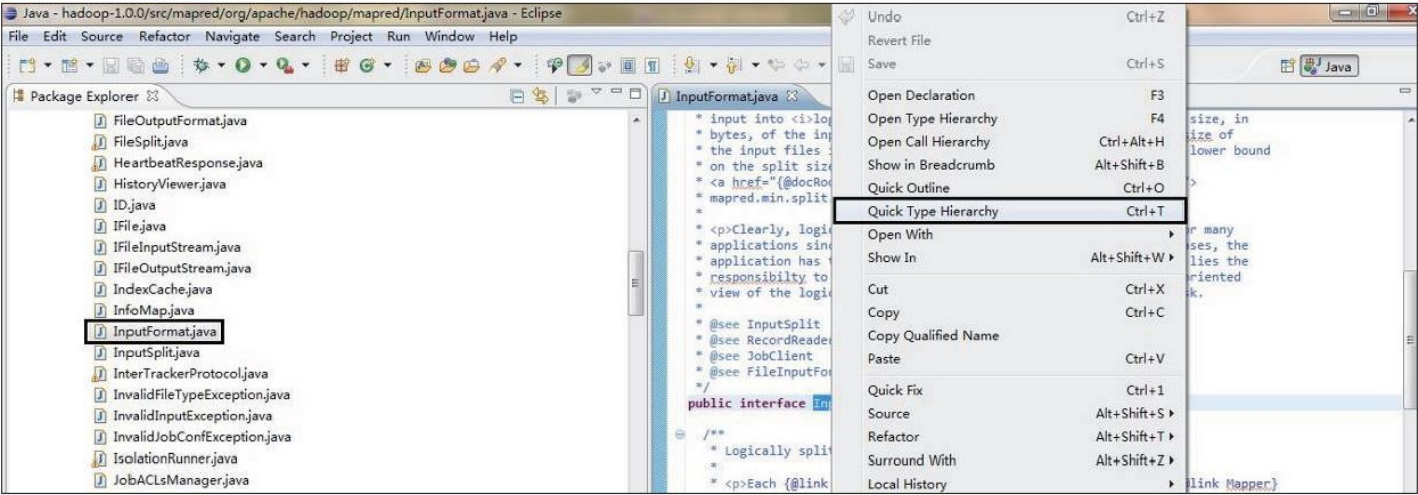


图 1-7 在Eclipse中查看Hadoop源代码中接口InputFormat的所有实现类

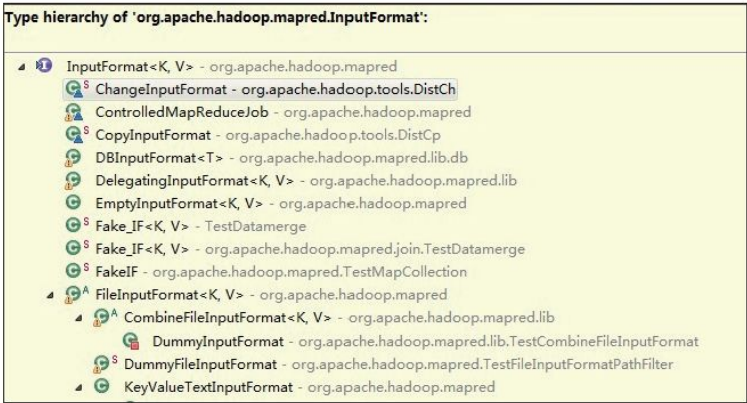


图 1-8 Eclipse列出接口InputFormat的所有实现类

2.查看函数的调用关系

在Eclipse中，选中某个方法名称，右击，在弹出的快捷菜单中选择“Open Call Hierarchy”，可在窗口“Call Hierarchy”中看到所有调用该方法的函数。

例如，如图1-9所示，打开src\mapred目录下org.apache.hadoop.mapred包中的JobTracker.java文件，查看调用方法initJob的所有函数，结果如图1-10所示。

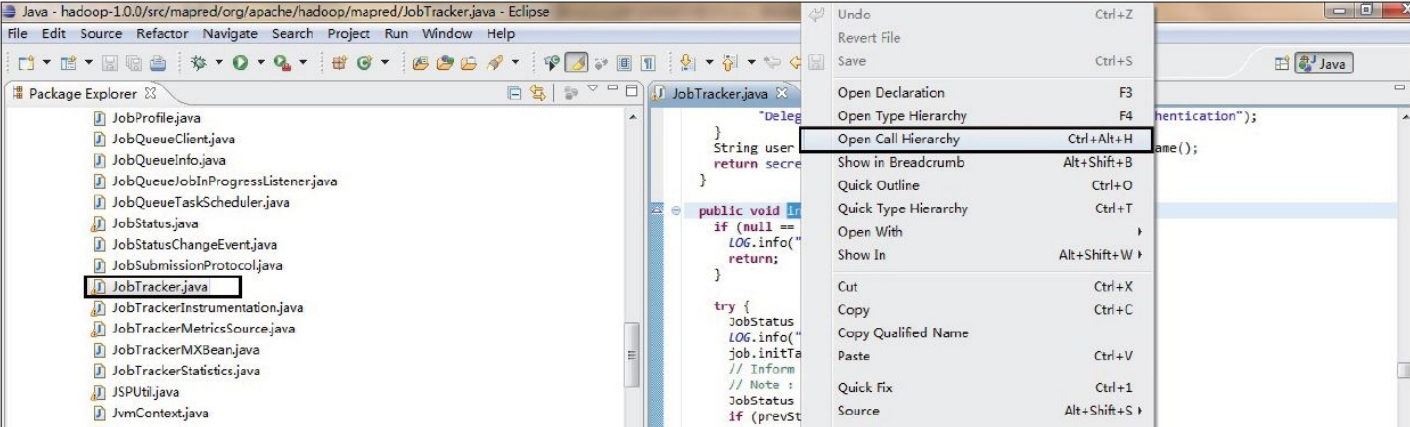


图 1-9 在Eclipse中查看Hadoop源代码中所有调用JobTracker.java中initJob方法的函数

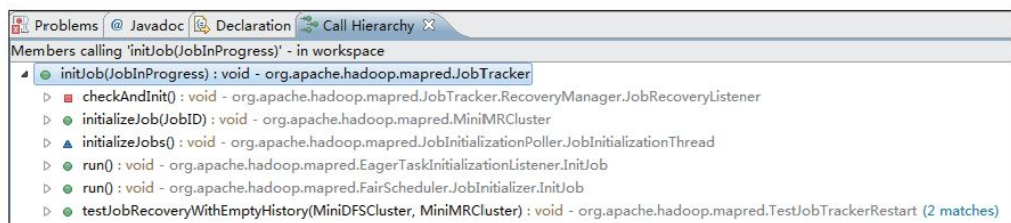


图 1-10 Eclipse列出所有调用initJob方法的函数

3.快速查找类对象的相关信息

同前两个小节类似，选中类对象，右击，在弹出的快捷菜单中选择“Open Declaration”，可跳转到类定义；选择“Quick Outline”，可查看类所有的成员变量和成员方法。具体细节本书不做详细介绍，读者可自行尝试。

1.4 Hadoop源代码组织结构

直接解压Hadoop压缩包后，可看到图1-11所示的目录结构，其中，比较重要的目录有src、conf、lib、bin等。下面分别介绍这几个目录的作用：

❑src: Hadoop源代码所在的目录。最核心的代码所在子目录分别是core、hdfs和mapred，它们分别实现了Hadoop最重要的三个模块，即基础公共库、HDFS实现和MapReduce实现。

❑conf: 配置文件所在目录。Hadoop的配置文件比较多，其设计原则可概括为如下两点。

○尽可能模块化，即每个重要模块拥有自己的配置文件，这样使得维护以及管理变得简单。

○动静分离，即将可动态加载的配置选项剥离出来，组成独立配置文件。比如，Hadoop 1.0.0版本之前，作业队列权限管理相关的配置选项被放在配置文件mapred-site.xml中，而该文件是不可以动态加载的，每次修改后必须重启MapReduce。但从1.0.0版本开始，这些配置选项被剥离放到独立配置文件mapred-queue-acls.xml中，该文件可以通过Hadoop命令行动态加载。conf目录下最重要的配置文件有core-site.xml、hdfs-site.xml和mapred-site.xml，分别设置了基础公共库core、分布式文件系统HDFS和分布式计算框架MapReduce的配置选项。

❑lib: Hadoop运行时依赖的三方库，包括编译好的jar包以及其他语言生成的动态库。Hadoop启动或者用户提交作业时，会自动加载这些库。

❑bin: 运行以及管理Hadoop集群相关的脚本。这里介绍几个常用的脚本。

○hadoop: 最基本且功能最完备的管理脚本，其他大部分脚本都会调用该脚本。

○start-all.sh/stop-all.sh: 启动/停止所有节点上的HDFS和MapReduce相关服务。

○start-mapred.sh/stop-mapred.sh: 单独启动/停止MapReduce相关服务。

○start-dfs.sh/stop-dfs.sh: 单独启动/停止HDFS相关服务。

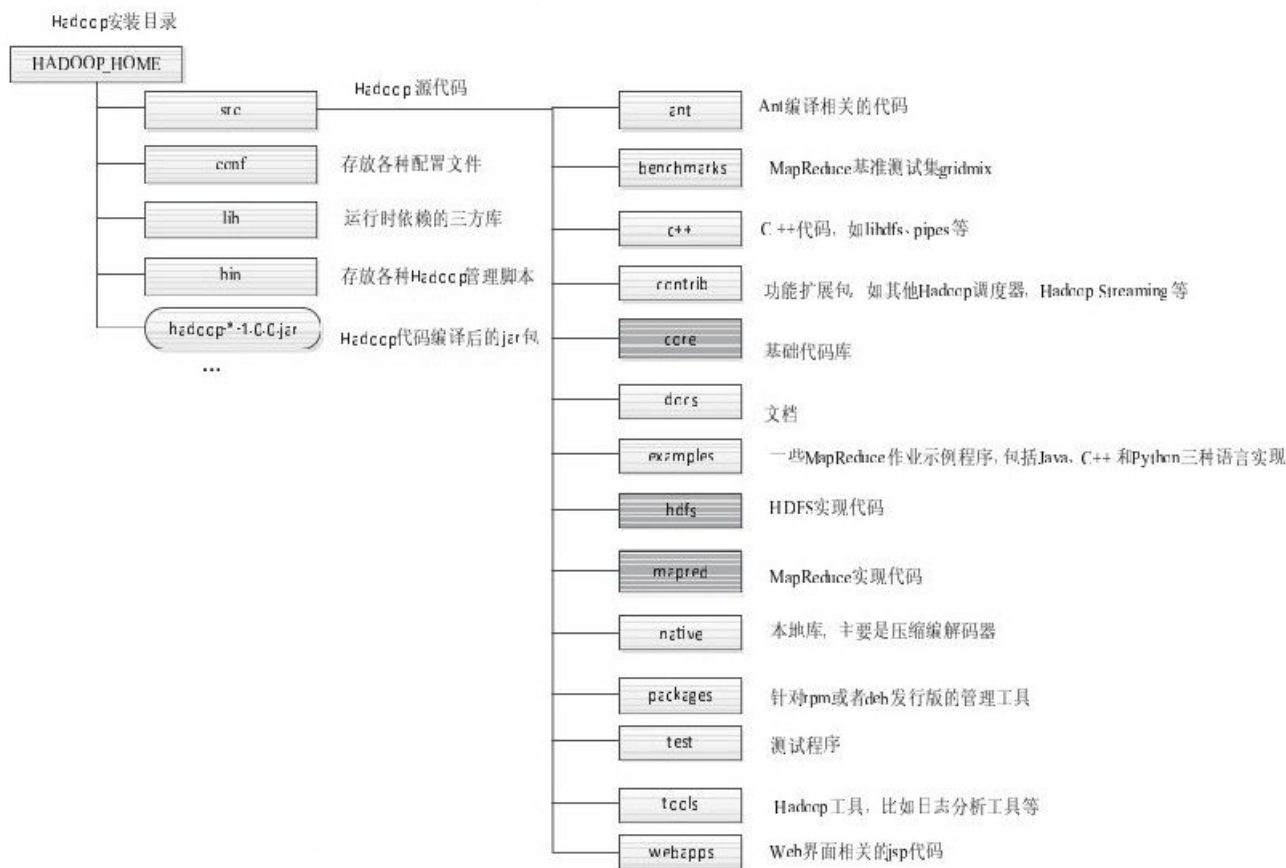


图 1-11 Hadoop安装目录结构

本书重点介绍MapReduce的实现原理，下面就Hadoop MapReduce源代码组织结构进行介绍。Hadoop MapReduce源代码组织结构^[1]如图1-12所示。

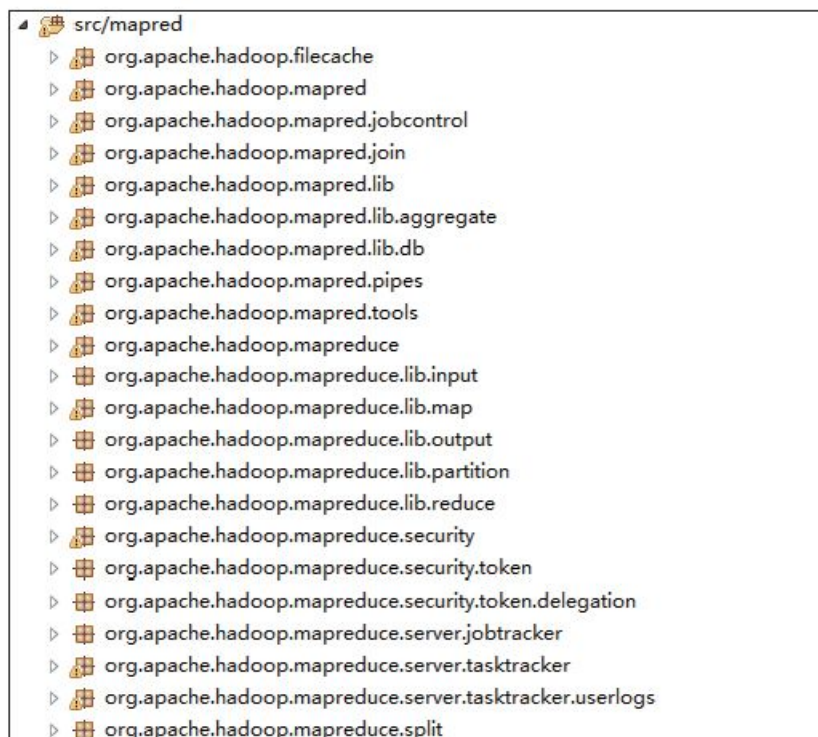


图 1-12 Hadoop MapReduce源代码组织结构

总体上看，Hadoop MapReduce分为两部分：一部分是org.apache.hadoop.mapred.*，这里面主要包含旧的对外编程接口以及MapReduce各个服务（JobTracker以及TaskTracker）的实现；另一部分是org.apache.hadoop.mapreduce.*，主要内容涉及新版本的对

外编程接口以及一些新特性（比如MapReduce安全）。

1.MapReduce编程模型相关

□org.apache.hadoop.mapred.lib.*: 这一系列Java包提供了各种可直接在应用程序中使用的InputFormat、Mapper、Partitioner、Reducer和OutputFormat，以减少用户编写MapReduce程序的工作量。

□org.apache.hadoop.mapred.jobcontrol: 该Java包允许用户管理具有相互依赖关系的作业（DAG作业）。

□org.apache.hadoop.mapred.join: 该Java包实现了map-side join算法^[2]。该算法要求数据已经按照key排好序，且分好片，这样可以只使用Map Task实现join算法，避免re-partition、sort、shuffling等开销。

□org.apache.hadoop.mapred.pipes: 该Java包允许用户用C/C++编写MapReduce作业。

□org.apache.hadoop.mapreduce: 该Java包定义了一套新版本的编程接口，这套接口比旧版接口封装性更好。

□org.apache.hadoop.mapreduce.*: 这一系列Java包根据新版接口实现了各种InputFormat、Mapper、Partitioner、Reducer和OutputFormat。

2.MapReduce计算框架相关

□org.apache.hadoop.mapred: Hadoop MapReduce最核心的实现代码，包括各个服务的具体实现。

□org.apache.hadoop.mapred.filecache: Hadoop DistributedCache实现。DistributedCache是Hadoop提供的文件分发工具，可将用户应用程序中需要的文件分发到各个节点上。

□org.apache.hadoop.mapred.tools: 管理控制Hadoop MapReduce，当前功能仅包括允许用户动态更新服务级别的授权策略和ACL（访问权限控制）属性。

□org.apache.hadoop.mapreduce.split: 该Java包的主要功能是根据作业的InputFormat生成相应的输入split。

□org.apache.hadoop.mapreduce.server.jobtracker: 该Java包维护了JobTracker可看到的TaskTracker状态信息和资源使用情况。

□org.apache.hadoop.mapreduce.server.tasktracker.*: TaskTracker的一些辅助类。

3.MapReduce安全机制相关

这里只涉及org.apache.hadoop.mapreduce.security.*。这一系列Java包实现了MapReduce安全机制。

[1] 不同版本Hadoop的源代码结构稍有差距，本书的分析是基于Hadoop 1.0.0版本的。

[2] Join算法是将两个表或者文件按照某个key值合并起来，在Hadoop中，可以在Map或者Reduce端进行合并。若在Reduce端进行合并，则需要进行re-partition、sort、shuffling等操作，开销很大。

1.5 Hadoop初体验

一般而言，我们想要深入学习一个新的系统时，首先要尝试使用该系统，了解系统对外提供的功能，然后通过某个功能逐步深入其实现细节。本节将介绍如何在伪分布式工作模式^[1]下使用Hadoop，包括启动Hadoop、访问HDFS以及向MapReduce提交作业等最基本的操作。本节只是有代表性地介绍Hadoop的一些基本使用方法，使读者对Hadoop有一个初步认识，并引导读者逐步进行更全面的学习。

1.5.1 启动Hadoop

步骤1 修改Hadoop配置文件。在conf目录下，修改mapred-site.xml、core-site.xml和hdfs-site.xml三个文件，在<configuration>与</configuration>之间添加以下内容。

❑ mapred-site.xml:

```
<property>
<name>mapred.job.tracker</name>
<value>localhost: 9001</value>
</property>
```

❑ core-site.xml:

```
<property>
<name>fs.default.name</name>
<value>hdfs: //localhost: 9000</value>
</property>
```

❑ hdfs-site.xml:

```
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
<property>
<name>dfs.permissions</name>
<value>>false</value>
</property>
```

如果是Windows环境，还需要在hadoop-env.xml中添加以下配置：

```
export JAVA_HOME=D:/hadoop/Java/jdk1.6.0_27
```

步骤2 设置免密码登录。前面提到Hadoop启动启动/停止脚本时需要通过SSH发送命令启动相关守护进程，为了避免每次启动/停止Hadoop输入密码进行验证，需设置免密码登录，设置步骤如下。

1) 打开命令行终端（Windows下为Cygwin终端，Linux下为Shell终端，下同），输入以下命令：

```
ssh-keygen-t rsa
```

执行上述命令后，将会在“~/.ssh”目录下生成公钥文件id_rsa.pub和私钥文件id_rsa。

2) 将公钥文件id_rsa.pub中的内容复制到相同目录下的authorized_keys文件中：

```
cd ~/.ssh/
cat id_rsa.pub >> authorized_keys
```

步骤3 启动Hadoop。在Hadoop安装目录中，按以下两步操作启动Hadoop。

1) 格式化HDFS:

```
bin/hadoop namenode-format
```

2) 启动Hadoop:

```
bin/start-all.sh
```

通过以下URL可查看MapReduce是否启动成功:

```
http://localhost: 50030/
```

通过以下URL可查看HDFS是否启动成功:

```
http://localhost: 50070/
```

经过以上两步操作，Hadoop成功启动，接下来可以通过Hadoop Shell或者Eclipse插件访问HDFS和提交MapReduce作业。下面两小节分别介绍Hadoop Shell和Eclipse插件的使用方法。

[1] 单机环境中，Hadoop有两种工作模式：本地模式和伪分布式模式。其中，本地模式完全运行在本地，不会加载任何MapReduce服务，因而不会涉及MapReduce最核心的代码实现；伪分布式模式即为“单点集群”，在该模式下，所有的守护进程均会运行在单个节点上，因而本节选用该工作模式。

1.5.2 Hadoop Shell介绍

在1.4节中曾提到，`bin`目录下的Hadoop脚本是最基础的集群管理脚本，用户可以通过该脚本完成各种功能，如HDFS文件管理、MapReduce作业管理等。该脚本的使用方法为：

```
hadoop [--config confdir] COMMAND
```

其中，`--config`用于设置Hadoop配置文件目录，默认目录为`${HADOOP_HOME}/conf`。而`COMMAND`是具体的某个命令，常用的有HDFS管理命令`fs`、作业管理命令`job`和作业提交命令`jar`等。它们的使用方法如下：

（1）HDFS管理命令`fs`和作业管理命令`job`

它们的用法一样，均为：

```
bin/hadoop command[genericOptions] [commandOptions]
```

其中，`command`可以是`fs`或者`job`，`genericOptions`是一些通用选项，`commandOptions`是`fs`或者`job`附加的命令选项。看下面两个例子。

❑在HDFS上创建一个目录`/test`：

```
bin/hadoop fs-mkdir/test
```

❑显示Hadoop上正在运行的所有作业：

```
bin/hadoop job-list
```

（2）作业提交命令`jar`

这个命令的用法是：

```
hadoop jar<jar> [mainClass] args..
```

其中，`<jar>`表示`jar`包名；`mainClass`表示`main class`名称，可以不必输入而由`jar`命令自动搜索；`args`是`main class`输入参数。举例如下：

```
bin/hadoop jar hadoop-examples-1.0.0.jar wordcount/test/input/test/output
```

其中，`wordcount`是`hadoop-examples-1.0.0.jar`中一个作业名称。顾名思义，该作业用于统计输入文件中的每个单词出现的次数，它有两个输入参数：输入数据目录（`/test/input`）和输出数据目录（`/test/output`）。

至于其他更多命令，读者可自行查阅Hadoop官方设计文档。

1.5.3 Hadoop Eclipse插件介绍

Hadoop提供了一个Eclipse插件以方便用户在Eclipse集成开发环境中使用Hadoop，如管理HDFS上的文件、提交作业、调试MapReduce程序等。本小节将介绍如何使用该插件访问HDFS、运行MapReduce作业和跟踪MapReduce作业运行过程。

1.编译生成Eclipse插件

用户需要自己生成Eclipse插件。Hadoop-1.0.0生成的jar包不能直接使用，需要进行部分修改，具体参考附录A中的问题2，其代码位于Hadoop安装目录的src/contrib/eclipse-plugin下，在该目录下，输入以下命令生成Eclipse插件：

```
ant-Declipse.home=/home/dong/eclipse-Dversion=1.0.0
```

其中，eclipse.home用来指定Eclipse安装目录，version是Hadoop版本号。\${HADOOP_HOME}/build/contrib目录下生成的hadoop-eclipse-plugin-1.0.0.jar文件即为Eclipse插件。

2.配置Eclipse插件

将生成的Eclipse插件hadoop-eclipse-plugin-1.0.0.jar复制到Eclipse安装目录的plugins文件夹下，然后重启Eclipse。

进入Eclipse后，按照以下步骤进行设置：在菜单栏中依次单击“Window”→“Show View”→“Other.....”，在对话框中依次单击“MapReduce Tools”→“Map/Reduce Locations”，会弹出图1-13a所示的对话框，按图中提示填写内容。

经上述步骤后，回到主界面，如图1-13b所示，可在“Project Explore”视图中查看分布式文件系统的内容，说明Eclipse插件安装成功。

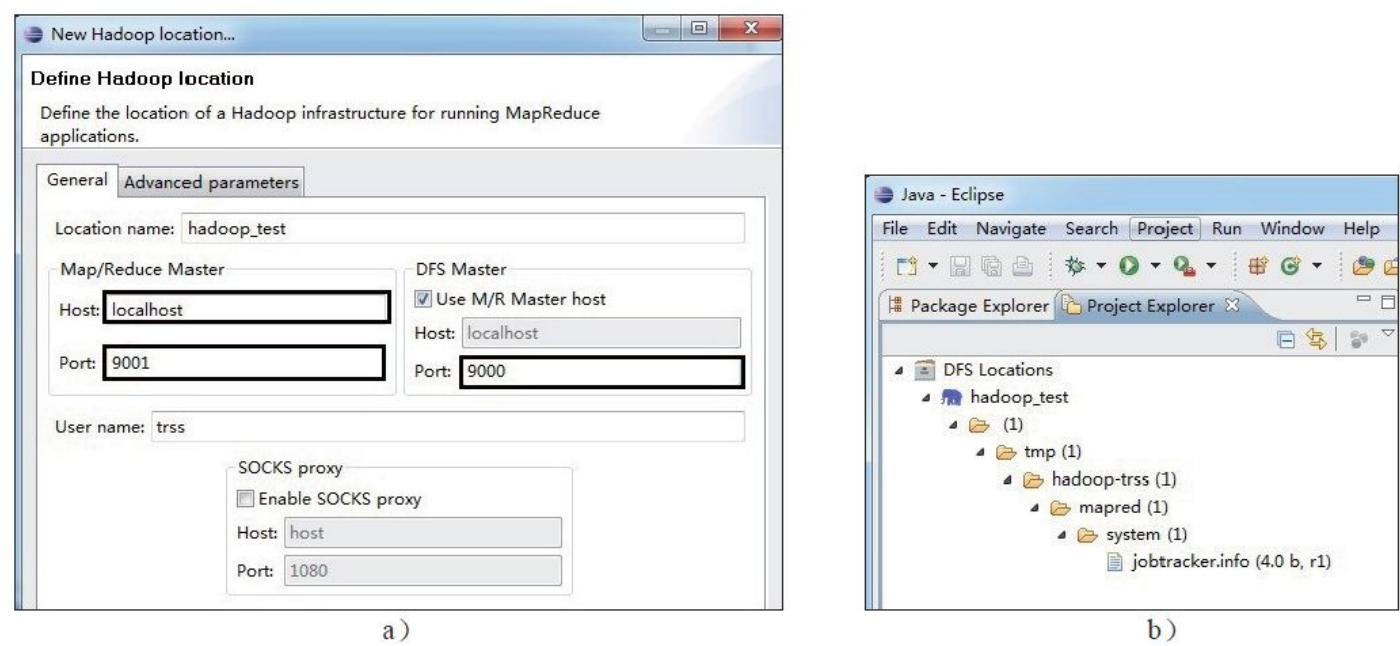


图 1-13 配置Hadoop Eclipse插件
a) 配置MapReduce和HDFS的主机名和端口号 b) 显示HDFS中的文件列表

3.运行MapReduce作业

前面提到，在伪分布式环境下，单个节点上会同时运行多种Hadoop服务。为了跟踪这些服务的运行轨迹，我们采用了以下方法：向Hadoop提交一个MapReduce作业，通过跟踪该作业的运行轨迹来分析Hadoop的内部实现原理。

该方法可通过以下三个步骤完成：

步骤1 新建一个MapReduce工程。在菜单栏中，依次单击“New”→“Other...”→“MapReduce Project”，会弹出图1-14所示的对话框。在该对话框中填写项目名称，并配置Hadoop安装目录，此处可直接选择前面已经建好的Java工程“hadoop-1.0.0”。

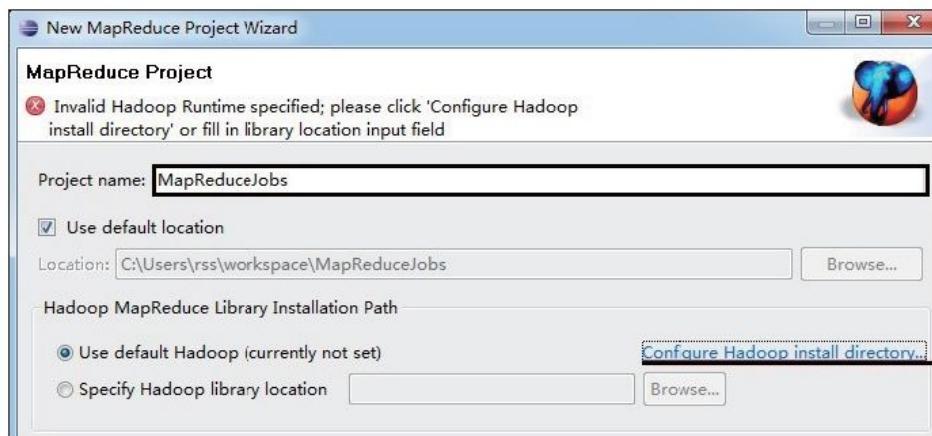


图 1-14 在Eclipse中创建MapReduce工程

步骤2 准备MapReduce作业。可直接将Hadoop源代码中src\examples\org\apache\hadoop\examples目录下的WordCount.java复制到新建的MapReduceJobs工程中。

步骤3 运行作业。

1) 准备数据。如图1-15所示，在HDFS上创建目录/test/input，并上传几个文本文件到该目录中。

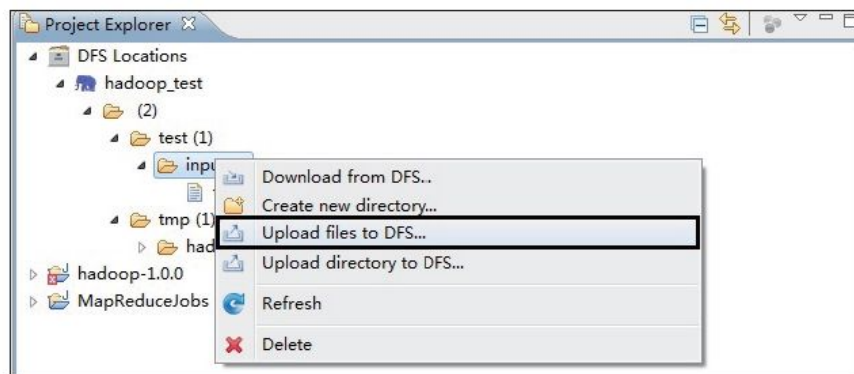


图 1-15 使用Eclipse插件上传文件到HDFS

2) 配置输入/输出路径。如图1-16所示，在WordCount.java中右击，在弹出的快捷菜单中依次单击“Run As”→“Run Configurations.....”，会出现图1-17所示的对话框。双击“Java Applications”选项，在新建的对话框中输入作业的输入/输出路径（中间用空格分隔），并单击“Apply”按钮保存。

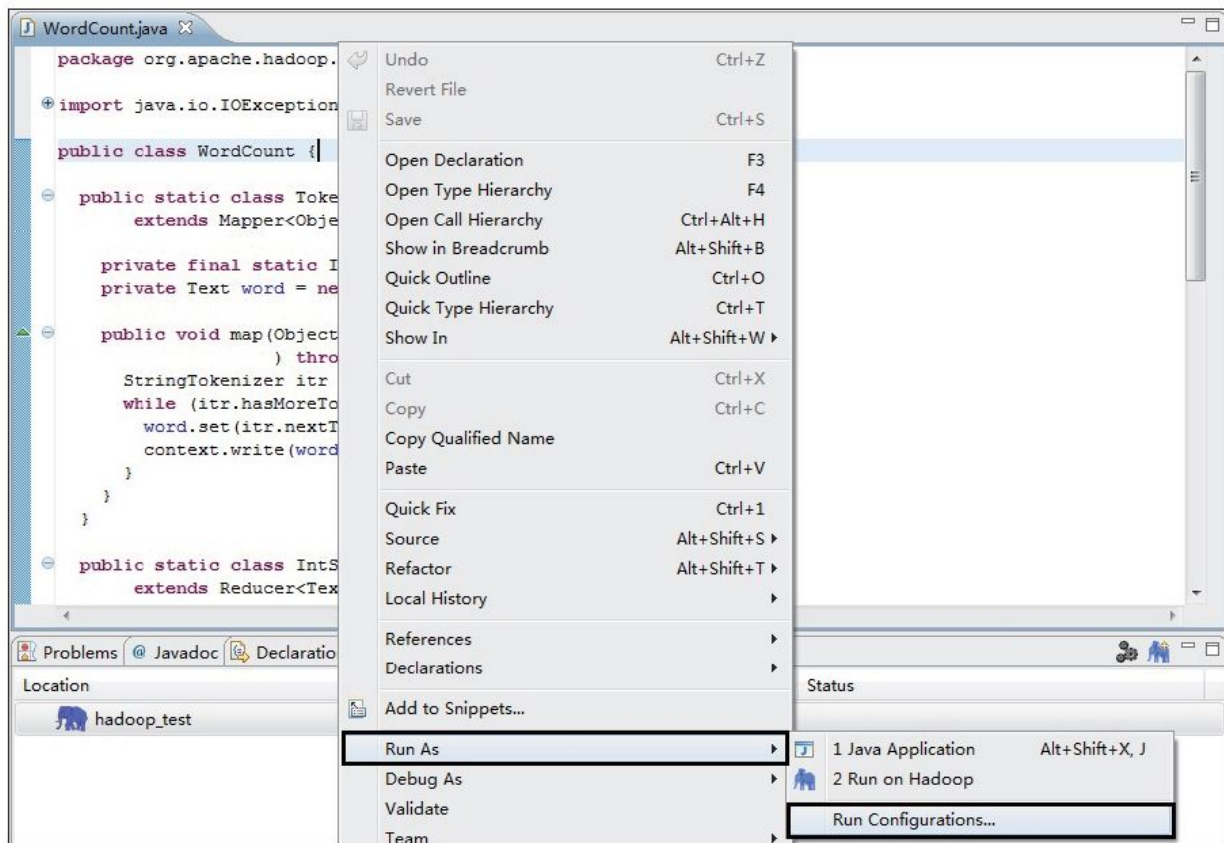


图 1-16 在Eclipse中配置作业的输入/输出路径（1）

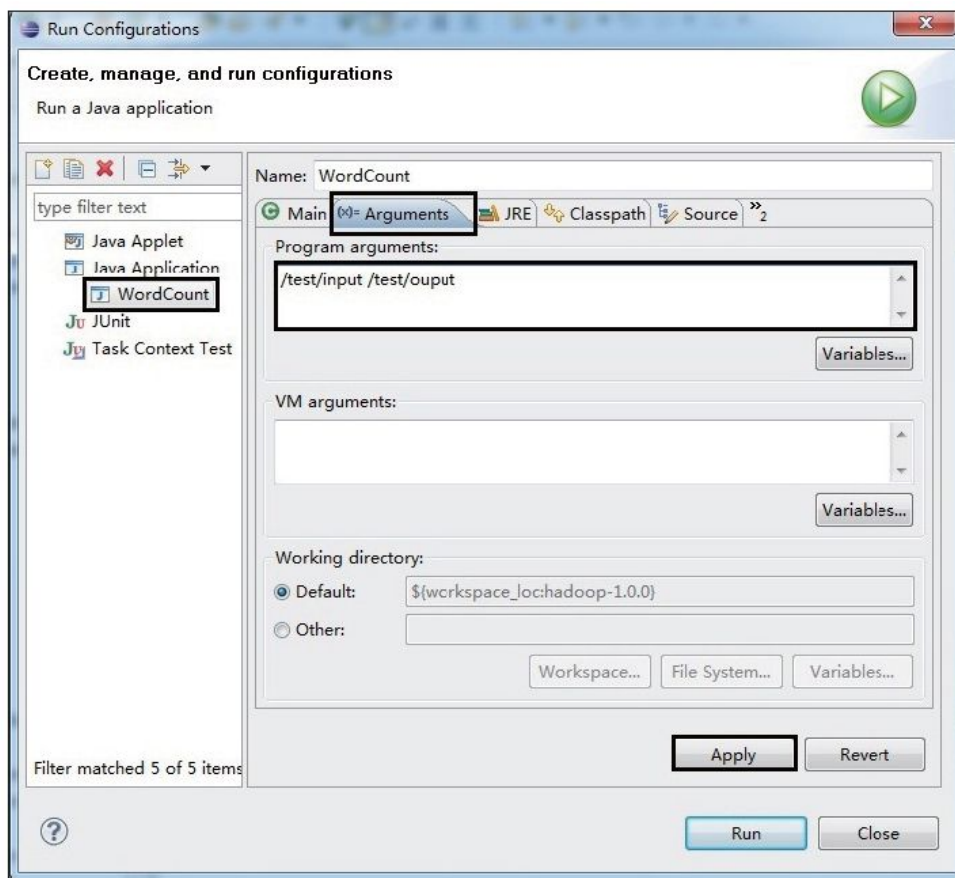


图 1-17 在Eclipse中配置作业的输入/输出路径（2）

3) 运行作业。在WordCount.java中右击，在弹出的快捷菜单中依次单击“Run As”→“Run on Hadoop”，会出现如图1-18所示的对话框。按图中的提示选择后，单击“Finish”按钮，作业开始运行。

此外，有兴趣的读者可以在MapReduce作业中设置断点，对作业进行断点调试。

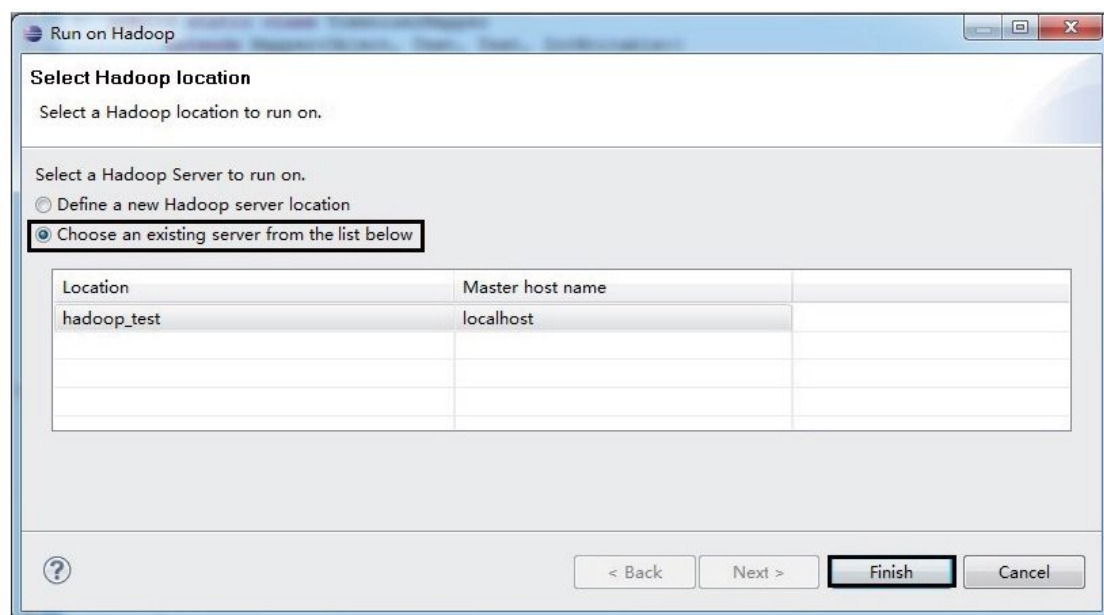


图 1-18 在Eclipse中运行作业

1.6 编译及调试Hadoop源代码

读者在阅读源代码的过程中，可能需要修改部分源代码或者使用调试工具以便跟踪某些变量值的变化过程，此时要用到Hadoop源代码编译和调试方法。本节将介绍Hadoop在伪分布式模式下的编译和调试方法，其中，调试方法主要介绍使用Eclipse远程调试和打印调试日志两种。

1.6.1 编译Hadoop源代码

在Windows或Linux环境下，打开命令行终端，转到Hadoop安装目录下并输入以下命令：

```
ant-Dversion=1.0.0{target}
```

其中，{target}值如表1-2所示，不同的target可对应生成不同的jar包，如：

```
ant-Dversion=1.0.0 examples
```

可生成hadoop-examples-1.0.0.jar，产生的jar包位于Hadoop安装目录的build文件夹下。

表 1-2 编译 target 与对应生成的 jar 包

target	jar 包
jar	hadoop-core-1.0.0.jar
examples	hadoop-examples-1.0.0.jar
tools-jar	hadoop-tools-1.0.0.jar
jar-test	hadoop-test-1.0.0.jar
ant-tasks	hadoop-ant-1.0.0.jar

1.6.2 调试Hadoop源代码

本小节介绍两种调试方式：利用Eclipse远程调试和打印调试日志。这两种方式均可以调试伪分布式工作模式和完全分布式工作模式下的Hadoop。本小节主要介绍伪分布式工作模式下的Hadoop调试方法。

1.利用Eclipse进行远程调试

下面以调试JobTracker为例，介绍利用Eclipse进行远程调试的基本方法。调试过程可分三步进行：

步骤1 调试模式下启动Hadoop。

在Hadoop安装目录下运行内容如下的Shell脚本：

```
export HADOOP_JOBTRACKER_OPTS="-Xdebug-Xrunjdpw: transport=dt_socket, address=8788, server=y, suspend=y"
bin/start-all.sh
```

如果脚本运行成功，则可以看到Shell命令行终端显示如下信息：

```
Listening for transport dt_socket at address: 8788
```

此时表明JobTracker处于监听状态。JobTracker将一直处于监听状态，直到收到debug确认信息。

步骤2 设置断点。

在前面新建的Java工程“hadoop-1.0.0”中，找到JobTracker相关代码，并在感兴趣的地方设置一些断点。

步骤3 在Eclipse中调试Hadoop程序。

在Eclipse的菜单栏中，依次单击“Run”→“Debug Configurations”→“Remote Java Applications”，打开图1-19所示的对话框，按图中的提示填写名称、JobTracker所在的host以及监听端口，并选择Hadoop源代码工程，进入图1-20所示的调试模式。

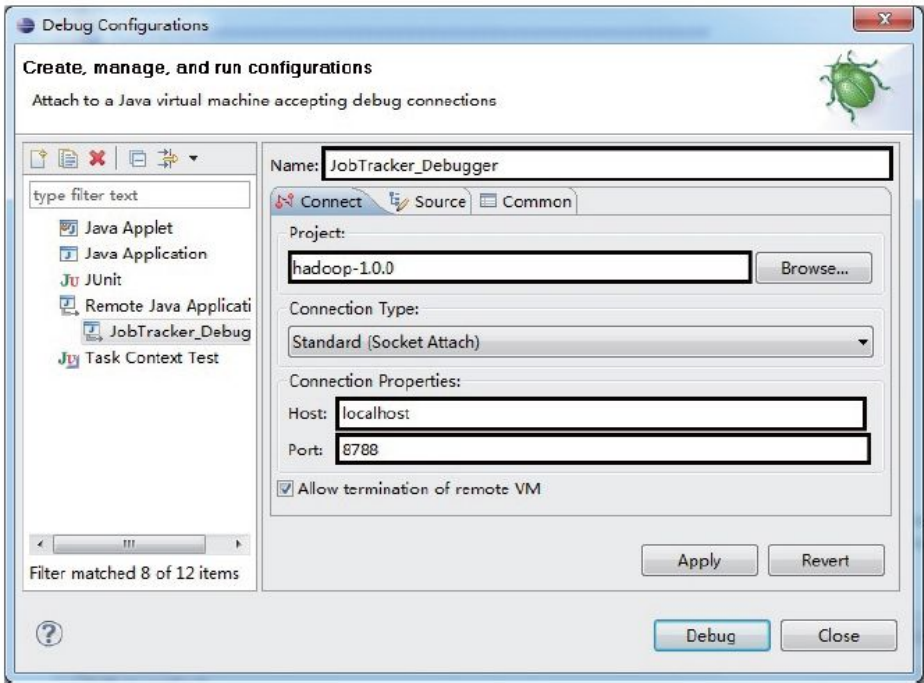


图 1-19 在Eclipse中配置远程调试器

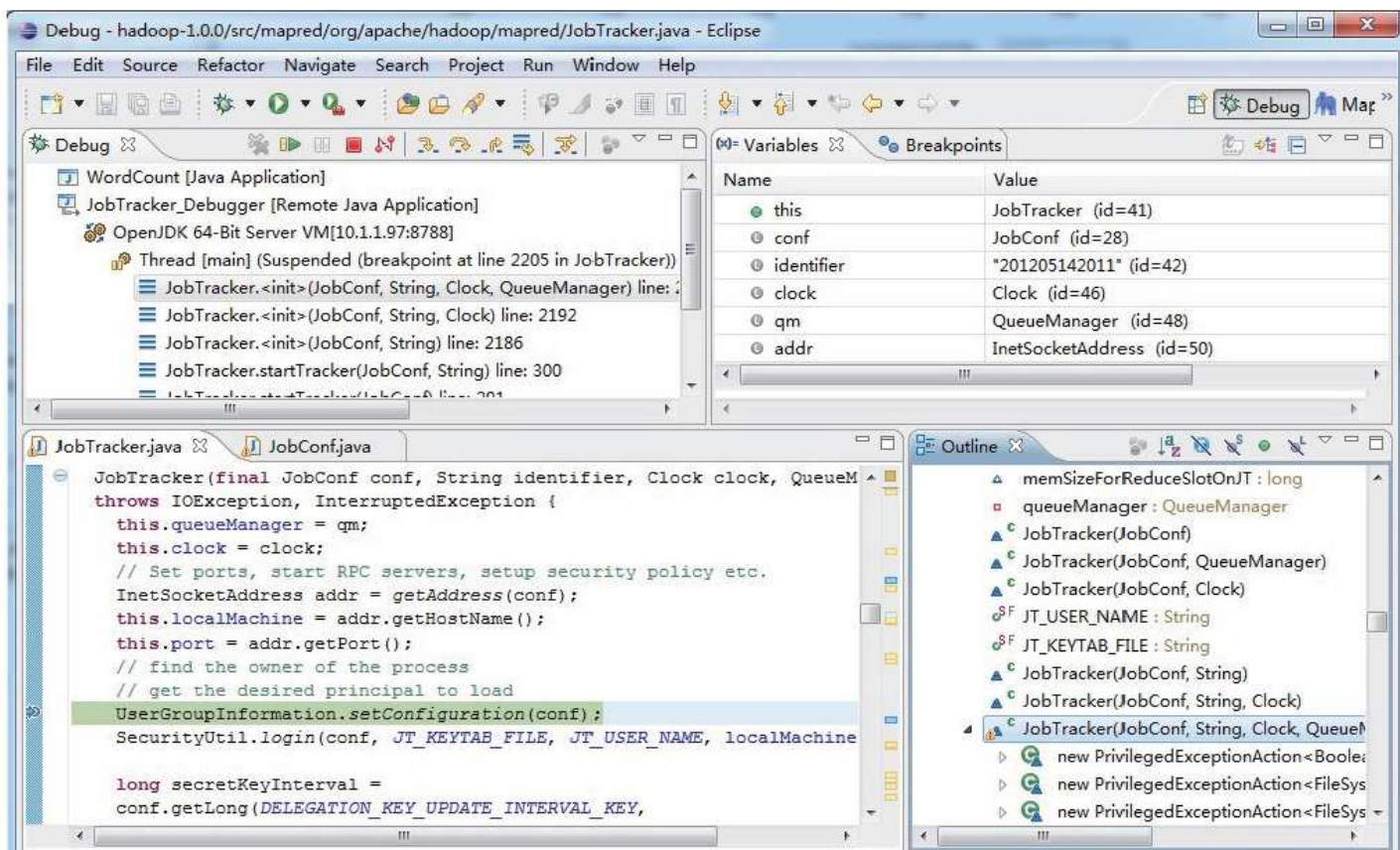


图 1-20 Eclipse中显示的Hadoop调试窗口

调试过程中，JobTracker输出的信息被存储到日志文件夹下的hadoop-XXX-jobtracker-localhost.log文件（XXX为当前用户名）中，可通过以下命令查看调试过程中打印的日志：

```
tail -f logs/hadoop-XXX-jobtracker-localhost.log
```

2.打印Hadoop调试日志

Hadoop使用了Apache log4j^[1]作为基础日志库。该日志库将日志分为5个级别，分别为DEBUG、INFO、WARN、ERROR和FATAL。这5个级别对应的日志信息重要程度不同，它们的重要程度由低到高依次为DEBUG<INFO<WARN<ERROR<FATAL。日志输出规则为：只输出级别不低于设定级别的日志信息。比如，级别设定为INFO，则INFO、WARN、ERROR和FATAL级别的日志信息都会被输出，但级别比INFO低的DEBUG则不会被输出。

在Hadoop源代码中，大部分Java文件中存在调试日志（DEBUG级别日志），但默认情况下，日志级别是INFO。为了查看更详细的运行状态，可采用以下几种方法打开DEBUG日志。

（1）使用Hadoop Shell命令

可使用Hadoop脚本中的daemonlog命令查看和修改某个类的日志级别，比如，可通过以下命令查看TaskTracker类的日志级别：

```
bin/hadoop daemonlog-getlevel${tasktracker-host}: 50075\
org.apache.hadoop.mapred.TaskTracker
```

可通过以下命令将JobTracker类的日志级别修改为DEBUG：

```
bin/hadoop daemonlog-setlevel${tasktracker-host}: 50075\
org.apache.hadoop.mapred.TaskTracker DEBUG
```

其中，tasktracker-host为TaskTracker的host，50075是TaskTracker的HTTP端口号（其他服务的HTTP端口号可参考附录B）。

（2）通过Web界面

用户可以通过Web界面查看和修改某个类的日志级别，比如，可通过以下URL修改TaskTracker类的日志级别：

```
http://${tasktracker-host}: 50075/logLevel
```

（3）修改log4j.properties文件

以上两种方法只能暂时修改日志级别。当Hadoop重启后会被重置，如果要永久性改变日志级别，可在目标节点配置目录下的log4j.properties文件中添加以下配置选项：

```
log4j.logger.org.apache.hadoop.mapred.TaskTracker=DEBUG
```

此外，有时为了专门调试某个Java文件，需要把该文件的相关日志输出到一个单独文件中，可在log4j.properties中添加以下内容：

```
#定义输出方式为自定义的TTOUT
log4j.logger.org.apache.hadoop.mapred.TaskTracker=DEBUG, TTOUT
#设置TTOUT的输出方式为输出到文件
log4j.appender.TTOUT=org.apache.log4j.FileAppender
#设置文件路径
log4j.appender.TTOUT.File=${hadoop.log.dir}/TaskTracker.log
#设置文件的布局
log4j.appender.TTOUT.layout=org.apache.log4j.PatternLayout
#设置文件的格式
log4j.appender.TTOUT.layout.ConversionPattern=%d{ISO8601}%p%c: %m%n
```

这些配置选项会把TaskTracker.java中的DEBUG日志写到日志目录下的TaskTracker.log文件中。

在阅读源代码的过程中，为了跟踪某个变量值的变化，读者可能需要自己添加一些DEBUG日志。在Hadoop源代码中，大部分类会定义一个日志打印对象。通过该对象，可打印各个级别的日志。比如，在JobTracker中由以下代码定义对象LOG：

```
public static final Log LOG=LogFactory.getLog (JobTracker.class);
```

用户可使用LOG对象打印调试日志，比如，可在JobTracker的main函数首行添加以下代码：

```
LOG.debug ("Start to lauch JobTracker.....");
```

然后重新编译Hadoop源代码，并将org.apache.hadoop.mapred.JobTracker的调试级别修改为DEBUG，重新启动Hadoop后便可以看到该调试信息。

[1] Apache log4j网址：<http://logging.apache.org/log4j/index.html>

1.7 小结

搭建一个高效的源代码学习环境是深入学习Hadoop的良好开端，本章主要内容正是帮助读者搭建一个这样的学习环境。在作者看来，一个高效的Hadoop学习环境至少应该包括源代码阅读环境、Hadoop使用环境和源代码编译调试环境，而本章正是围绕这三个环境的搭建方法编写的。

本章首先分别介绍了在Linux和Windows环境下搭建Hadoop源代码阅读环境的方法；在此基础上，进一步介绍了Hadoop的基本使用方法，主要涉及Hadoop Shell和Eclipse插件两种工具的使用；最后介绍了Hadoop源代码编译和调试方法，其中，调试方法主要介绍了使用Eclipse远程调试和打印调试日志两种。

第2章 MapReduce设计理念与基本架构

第1章介绍了Hadoop学习环境的搭建方法，这是学习Hadoop需要进行的最基本的准备工作。而在这一章中，我们将从设计理念和基本架构方面对Hadoop MapReduce进行介绍，同样，这属于准备工作的一部分。通过本章的介绍将会为后面几章深入剖析MapReduce内部实现奠定基础。

MapReduce是一个分布式计算框架，主要由两部分组成：编程模型和运行时环境。其中，编程模型为用户提供了非常易用的编程接口，用户只需要像编写串行程序一样实现几个简单的函数即可实现一个分布式程序，而其他比较复杂的工作，如节点间的通信、节点失效、数据切分等，全部由MapReduce运行时环境完成，用户无须关心这些细节。在本章中，我们将从设计目标、编程模型和基本架构等方面对MapReduce框架进行介绍。

2.1 Hadoop发展史

2.1.1 Hadoop产生背景

Hadoop最早起源于Nutch^[1]。Nutch是一个开源的网络搜索引擎，由Doug Cutting于2002年创建。Nutch的设计目标是构建一个大型的全网搜索引擎，包括网页抓取、索引、查询等功能，但随着抓取网页数量的增加，遇到了严重的可扩展性问题，即不能解决数十亿网页的存储和索引问题。之后，谷歌发表的两篇论文为该问题提供了可行的解决方案。一篇是2003年发表的关于谷歌分布式文件系统（GFS）的论文^[2]。该论文描述了谷歌搜索引擎网页相关数据的存储架构，该架构可解决Nutch遇到的网页抓取和索引过程中产生的超大文件存储需求的问题。但由于谷歌仅开源了思想而未开源代码，Nutch项目组便根据论文完成了一个开源实现，即Nutch的分布式文件系统（NDFS）。另一篇是2004年发表的关于谷歌分布式计算框架MapReduce的论文^[3]。该论文描述了谷歌内部最重要的分布式计算框架MapReduce的设计艺术，该框架可用于处理海量网页的索引问题。同样，由于谷歌未开源代码，Nutch的开发人员完成了一个开源实现。由于NDFS和MapReduce不仅适用于搜索领域，2006年年初，开发人员便将其移出Nutch，成为Lucene^[4]的一个子项目，称为Hadoop。大约同一时间，Doug Cutting加入雅虎公司，且公司同意组织一个专门的团队继续发展Hadoop。同年2月，Apache Hadoop项目正式启动以支持MapReduce和HDFS的独立发展。2008年1月，Hadoop成为Apache顶级项目，迎来了它的快速发展期。

[1] <http://nutch.apache.org/>

[2] 论文：Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, “The Google file system”, Proceedings of the nineteenth ACM Symposium on Operating Systems Principles.

[3] 论文：J.Dean and S.Ghemawat, “Mapreduce: simplified data processing on large clusters”, Proceedings of the 6th Conference on Symposium on Operating Systems Design& Implementation.

[4] <http://lucene.apache.org/>

2.1.2 Apache Hadoop新版本的特性

当前Apache Hadoop版本非常多，本小节将帮助读者梳理各个版本的特性以及它们之间的联系。在讲解Hadoop各版本之前，先要了解Apache软件发布方式。对于任何一个Apache开源项目，所有的基础特性均被添加到一个称为“trunk”的主代码线（main codeline）。当需要开发某个重要的特性时，会专门从主代码线中延伸出一个分支（branch），这被称为一个候选发布版（candidate release）。该分支将专注于开发该特性而不再添加其他新的特性，待基本bug修复之后，经过相关人士投票便会对外公开成为发布版（release version），并将该特性合并到主代码线中。需要注意的是，多个分支可能会同时进行研发，这样，版本高的分支可能先于版本低的分支发布。

由于Apache以特性为准延伸新的分支，故在介绍Apache Hadoop版本之前，先介绍几个独立产生的Apache Hadoop新版本的重大特性：

□ Append^[1]：HDFS Append主要完成追加文件内容的功能，也就是允许用户以append方式修改HDFS上的文件。HDFS最初的一个设计目标是支持MapReduce编程模型，而该模型只需要写一次文件，之后仅进行读操作而不会对其修改，即“write-once-read-many”，这就不需要支持文件追加功能。但随着HDFS变得流行，一些具有写需求的应用想以HDFS作为存储系统，比如，有些应用程序需要往HDFS上某个文件中追加日志信息，HBase需使用HDFS具有的append功能以防止数据丢失^[2]等。

□ HDFS RAID^[3]：Hadoop RAID模块在HDFS之上构建了一个新的分布式文件系统：Distributed Raid FileSystem（DRFS）。该系统采用了Erasure Codes增强对数据的保护。有了这样的保护，可以采用更少的副本数来保持同样的可用性保障，进而为用户节省大量存储空间。

□ Symlink^[4]：让HDFS支持符号链接。符号链接是一种特殊的文件，它以绝对或者相对路径的形式指向另外一个文件或者目录（目标文件）。当程序向符号链接中写数据时，相当于直接向目标文件中写数据。

□ Security^[5]：Hadoop的HDFS和MapReduce均缺乏相应的安全机制，比如在HDFS中，用户只要知道某个block的blockID，便可以绕过NameNode直接从DataNode上读取该block，用户可以向任意DataNode上写block；在MapReduce中，用户可以修改或者删掉任意其他用户的作业等。为了增强Hadoop的安全机制，从2009年起，Apache专门组成一个团队，为Hadoop增加基于Kerberos和Deletion Token的安全认证和授权机制。

□ MRv1：第一代MapReduce计算框架。它由两部分组成：编程模型（programmingmodel）和运行时环境（runtime environment）。它的基本编程模型是将问题抽象成Map和Reduce两个阶段。其中，Map阶段将输入数据解析成key/value，迭代调用map()函数处理后，再以key/value的形式输出到本地目录；Reduce阶段则将key相同的value进行规约处理，并将最终结果写到HDFS上。它的运行时环境由两类服务组成：JobTracker和TaskTracker，其中，JobTracker负责资源管理和所有作业的控制，而TaskTracker负责接收来自JobTracker的命令并执行它。

□ YARN/MRv2：针对MRv1中的MapReduce在扩展性和多框架支持方面的不足，提出了全新的资源管理框架YARN（Yet Another Resource Negotiator）。它将JobTracker中的资源管理和作业控制功能分开，分别由两个不同进程ResourceManager和ApplicationMaster实现。其中，ResourceManager负责所有应用程序的资源分配，而ApplicationMaster仅负责管理一个应用程序。

□ NameNode Federation^[6]：针对Hadoop 1.0中NameNode内存约束限制其扩展性问题提出的改进方案。它将NameNode横向扩展成多个，其中，每个NameNode分管一部分目录。这不仅增强了HDFS扩展性，也使HDFS NameNode具备了隔离性。

□ NameNode HA^[7]：HDFS NameNode存在两个问题，即NameNode内存约束限制扩展性和单点故障。其中，第一个问题通过NameNode Federation方案解决，而第二个问题则通过NameNode热备方案（即NameNode HA）实现。

[1] 0.20-append: <https://issues.apache.org/jira/browse/HDFS-200>, 0.21.0-append: <https://issues.apache.org/jira/browse/HDFS-265>

[2] <http://hbase.apache.org/book/hadoop.html>

[3] <http://wiki.apache.org/hadoop/HDFS-RAID>与<https://issues.apache.org/jira/browse/HDFS-503>

[4] <https://issues.apache.org/jira/browse/HDFS-245>

[5] <https://issues.apache.org/jira/browse/HADOOP-4487>

[6] <https://issues.apache.org/jira/browse/HDFS-1052>

[7] <https://issues.apache.org/jira/browse/HDFS-1623>

2.1.3 Hadoop版本变迁

到2012年5月为止，Apache Hadoop已经出现四个大的分支，如图2-1所示。

Apache Hadoop的四大分支构成了四个系列的Hadoop版本。

1.0.20.X系列

0.20.2版本发布后，几个重要的特性没有基于trunk而是在0.20.2基础上继续研发。值得一提的主要有两个特性：Append与Security。其中，含Security特性的分支以0.20.203版本发布，而后续的0.20.205版本综合了这两个特性。需要注意的是，之后的1.0.0版本仅是0.20.205版本的重命名。0.20.X系列版本是最令用户感到疑惑的，因为它们具有的一些特性，trunk上没有；反之，trunk上有一些特性，0.20.X系列版本却没有。

2.0.21.0/0.22.X系列

这一系列版本将整个Hadoop项目分割成三个独立的模块，分别是Common、HDFS和MapReduce。HDFS和MapReduce都对Common模块有依赖性，但是MapReduce对HDFS并没有依赖性。这样，MapReduce可以更容易地运行其他分布式文件系统，同时，模块间可以独立开发。具体各个模块的改进如下。

- Common模块：最大的新特性是在测试方面添加了Large-Scale Automated TestFramework [1] 和Fault Injection Framework [2]。
- HDFS模块：主要增加的新特性包括支持追加操作与建立符号连接、SecondaryNameNode改进（Secondary NameNode被剔除，取而代之的是Checkpoint Node，同时添加一个Backup Node的角色，作为NameNode的冷备）、允许用户自定义block放置算法等。
- MapReduce模块：在作业API方面，开始启动新MapReduce API，但老的API仍然兼容。

0.22.0在0.21.0的基础上修复了一些bug并进行了部分优化。

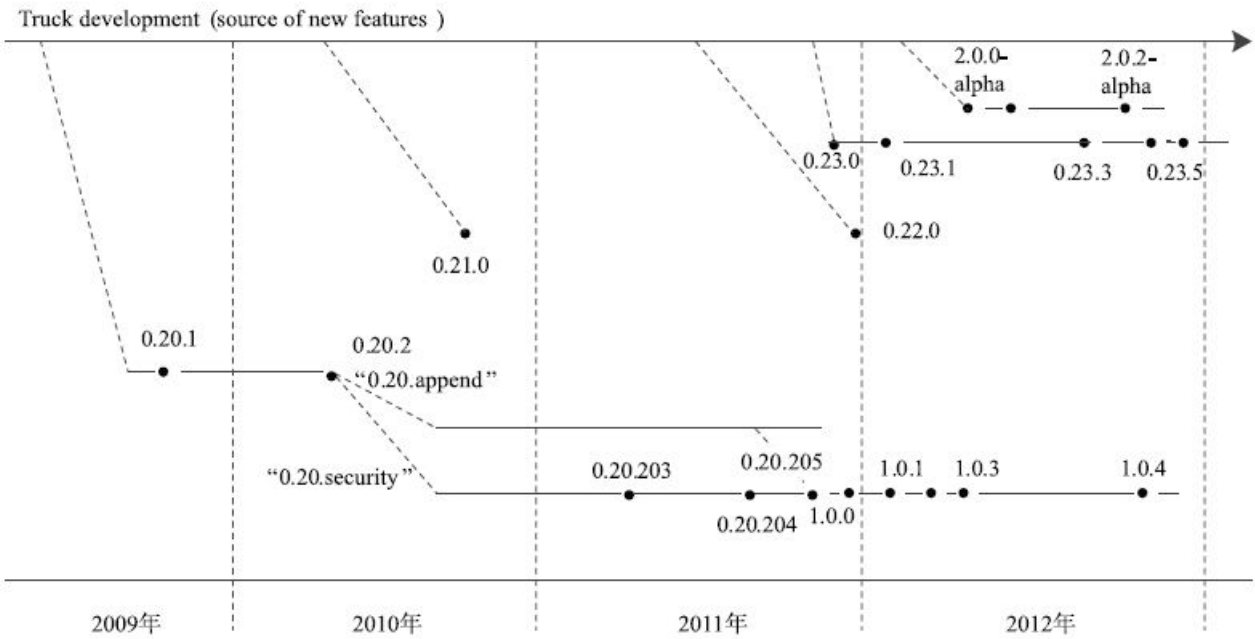


图 2-1 Hadoop版本变迁图 [3]

3.0.23.X系列

0.23.X是为了克服Hadoop在扩展性和框架通用性方面的不足而提出来的。它实际上是一个全新的平台，包括分布式文件系统HDFS Federation和资源管理框架YARN两部分，可对接入的各种计算框架（如MapReduce、Spark [4] 等）进行统一管理。它的发行版自带MapReduce库，而该库集成了迄今为止所有的MapReduce新特性。

4.2.X系列

同0.23.X系列一样，2.X系列也属于下一代Hadoop。与0.23.X系列相比，2.X系列增加了NameNode HA和Wire-compatibility等新特性。

表2-1总结了Hadoop各个发布版的特性以及稳定性。

表 2-1 Hadoop 各个发布版的特性以及稳定性

时间	发布版本	特性								是否稳定
		Append	RAID	Symlink	Security	MRv1	YARN	NameNode Federation	NameNode HA	
2010 年	0.20.2	×	×	×	×	√	×	×	×	是
	0.21.0	√	√	√	×	√	×	×	√	否
2011 年	0.20.203	×	×	×	√	√	×	×	×	是 (Yahoo ! 在 4 500 个节点上 部署)
	0.20.205 (1.0.0)	√	×	×	√	√	×	×	×	是
	0.22.0	√	√	√	√ [⊖]	√	×	×	√	否
	0.23.0	√	√	√	√	×	√	√	×	否
	1.X	√	×	×	√	√	×	×	×	是
2012 年	2.X	√	√	√	√	×	√	√	√	否

本书之所以以分析Apache Hadoop 1.0.0为主，主要是因为这是一个稳定的版本，再有其为1.0.0，具有里程碑意义。Apache发布这个版本，也是希望该版本成为业界的规范。需要注意的是，尽管本书以分析Apache Hadoop 1.0.0版本为主，但本书内容适用于所有Apache Hadoop 1.X版本 [6] 。

[1] 参考<https://issues.apache.org/jira/browse/HADOOP-6332>

[2] 参考http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework.html

[3] 图片修改自：<http://www.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/>

[4] Spark是一种内存计算框架，支持迭代式计算，主页是<http://www.spark-project.org/>.

[5] 0.22.0版本中只有HDFS Security，没有MapReduce Security。

[6] 不同版本之间细节可能稍有不同，此时以Hadoop 1.0.0版本为主。

2.2 Hadoop MapReduce设计目标

通过上一节关于Hadoop MapReduce历史的介绍我们知道，Hadoop MapReduce诞生于搜索领域，主要解决搜索引擎面临的海量数据处理扩展性差的问题。它的实现很大程度上借鉴了谷歌MapReduce的设计思想，包括简化编程接口、提高系统容错性等。总结Hadoop MapReduce设计目标，主要有以下几个。

□易于编程：传统的分布式程序设计（如MPI）非常复杂，用户需要关注的细节非常多，比如数据分片、数据传输、节点间通信等，因而设计分布式程序的门槛非常高。Hadoop的一个重要设计目标便是简化分布式程序设计，将所有并行程序均需要关注的设计细节抽象成公共模块并交由系统实现，而用户只需专注于自己的应用程序逻辑实现，这样简化了分布式程序设计且提高了开发效率。

□良好的扩展性：随着公司业务的发展，积累的数据量（如搜索公司的网页量）会越来越大，当数据量增加到一定程度后，现有的集群可能已经无法满足其计算能力和存储能力，这时候管理员可能期望通过添加机器以达到线性扩展集群能力的目的。

□高容错性：在分布式环境下，随着集群规模的增加，集群中的故障率（这里的“故障”包括磁盘损坏、机器宕机、节点间通信失败等硬件故障和坏数据或者用户程序bug产生的软件故障）会显著增加，进而导致任务失败和数据丢失的可能性增加。为此，Hadoop通过计算迁移或者数据迁移等策略提高集群的可用性与容错性。

2.3 MapReduce编程模型概述

2.3.1 MapReduce编程模型简介

从MapReduce自身的命名特点可以看出，MapReduce由两个阶段组成：Map和Reduce。用户只需编写map()和reduce()两个函数，即可完成简单的分布式程序的设计。

map()函数以key/value对作为输入，产生另外一系列key/value对作为中间输出写入本地磁盘。MapReduce框架会自动将这些中间数据按照key值进行聚集，且key值相同（用户可设定聚集策略，默认情况下是对key值进行哈希取模）的数据被统一交给reduce()函数处理。

reduce()函数以key及对应的value列表作为输入，经合并key相同的value值后，产生另外一系列key/value对作为最终输出写入HDFS。

下面以MapReduce中的“hello world”程序——WordCount为例介绍程序设计方法。

“hello world”程序是我们学习任何一门编程语言编写的第一个程序。它简单且易于理解，能够帮助读者快速入门。同样，分布式处理框架也有自己的“hello world”程序：WordCount。它完成的功能是统计输入文件中的每个单词出现的次数。在MapReduce中，可以这样编写（伪代码）。

其中Map部分如下：

```
//key: 字符串偏移量
//value: 一行字符串内容
map (String key, String value) :
//将字符串分割成单词
words=SplitIntoTokens (value);
for each word w in words:
EmitIntermediate (w, "1");
```

Reduce部分如下：

```
//key: 一个单词
//values: 该单词出现的次数列表
reduce (String key, Iterator values) :
int result=0;
for each v in values:
result+=StringToInt (v);
Emit (key, IntToString (result));
```

用户编写完MapReduce程序后，按照一定的规则指定程序的输入和输出目录，并提交到Hadoop集群中。作业在Hadoop中的执行过程如图2-2所示。Hadoop将输入数据切分成若干个输入分片（input split，后面简称split），并将每个split交给一个Map Task处理；Map Task不断地从对应的split中解析出一个个key/value，并调用map()函数处理，处理完之后根据Reduce Task个数将结果分成若干个分片（partition）写到本地磁盘；同时，每个Reduce Task从每个Map Task上读取属于自己的那个partition，然后使用基于排序的方法将key相同的数据聚集在一起，调用reduce()函数处理，并将结果输出到文件中。

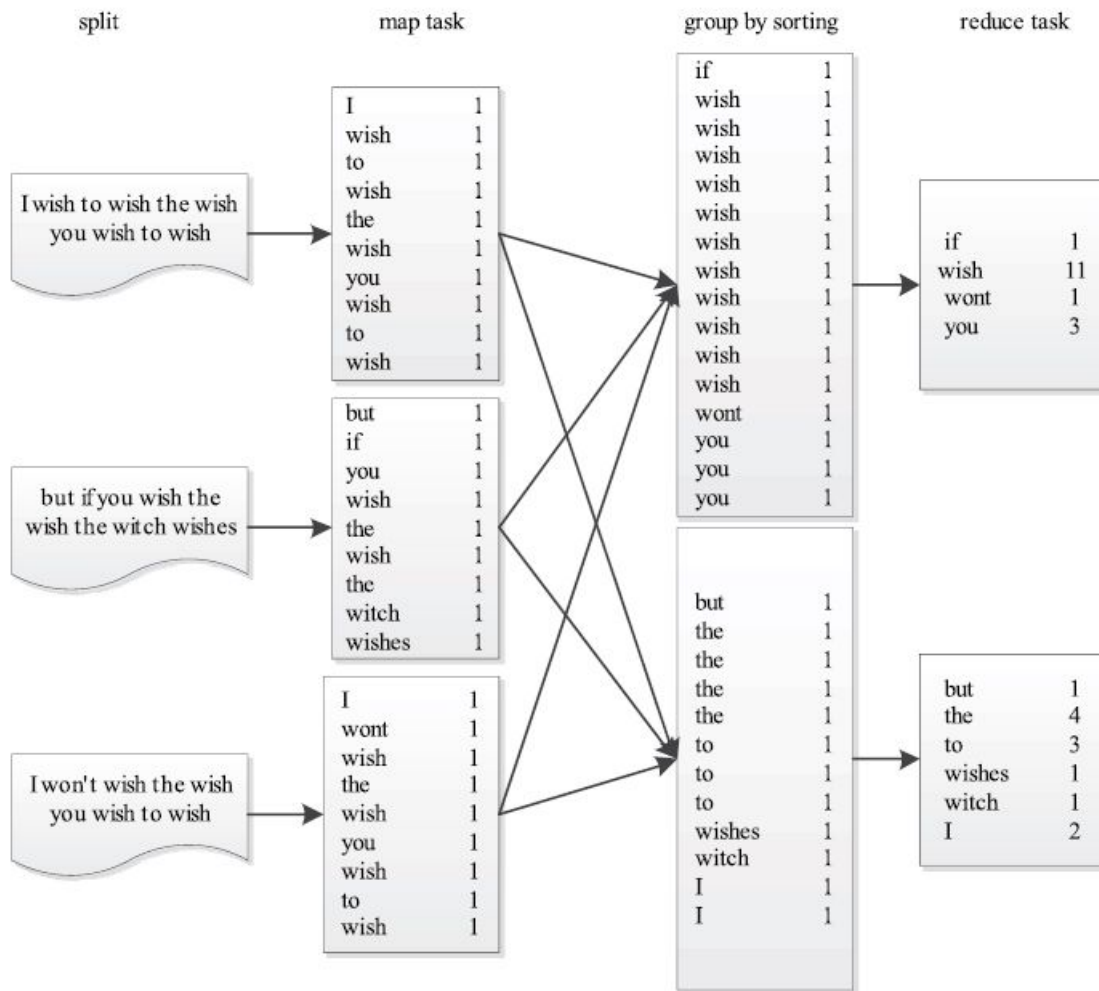


图 2-2 WordCount程序运行过程

细心的读者可能注意到，上面的程序还缺少三个基本的组件，功能分别是：①指定输入文件格式。将输入数据切分成若干个split，且将每个split中的数据解析成一个个map()函数要求的key/value对。②确定map()函数产生的每个key/value对发给哪个Reduce Task函数处理。③指定输出文件格式，即每个key/value对以何种形式保存到输出文件中。

在Hadoop MapReduce中，这三个组件分别是InputFormat、Partitioner和OutputFormat，它们均需要用户根据自己的应用需求配置。而对于上面的WordCount例子，默认情况下Hadoop采用的默认实现正好可以满足要求，因而不必再提供。

综上所述，Hadoop MapReduce对外提供了5个可编程组件，分别是InputFormat、Mapper、Partitioner、Reducer和OutputFormat^[1]。本书将在第3章中详细介绍它们的设计思路以及扩展实现。

^[1] 还有一个组件是Combiner，它通常用于优化MapReduce程序性能，但不属于必备组件。

2.3.2 MapReduce编程实例

MapReduce能够解决的问题有一个共同特点：任务可以被分解为多个子问题，且这些子问题相对独立，彼此之间不会有牵制，待并行处理完这些子问题后，任务便被解决。在实际应用中，这类问题非常庞大，谷歌在论文中提到了MapReduce的一些典型应用，包括分布式grep、URL访问频率统计、Web连接图反转、倒排索引构建、分布式排序等，这些均是比较简单的应用。下面介绍一些比较复杂的应用。

(1) Top K问题

在搜索引擎领域中，常常需要统计最近最热门的K个查询词，这就是典型的“Top K”问题，也就是从海量查询中统计出现频率最高的前K个。该问题可分解成两个MapReduce作业，分别完成统计词频和找出词频最高的前K个查询词的功能。这两个作业存在依赖关系，第二个作业需要依赖前一个作业的输出结果。第一个作业是典型的WordCount问题。对于第二个作业，首先map()函数中输出前K个频率最高的词，然后由reduce()函数汇总每个Map任务得到的前K个查询词，并输出频率最高的前K个查询词。

(2) K-means聚类

K-means是一种基于距离的聚类算法。它采用距离作为相似性的评价指标，认为两个对象的距离越近，其相似度就越大。该算法解决的问题可抽象成：给定正整数K和N个对象，如何将这些数据点划分为K个聚类？

该问题采用MapReduce计算的思路如下：首先随机选择K个对象作为初始中心点，然后不断迭代计算，直到满足终止条件（达到迭代次数上限或者数据点到中心点距离的平方和最小）。在第I轮迭代中，map()函数计算每个对象到中心点的距离，选择距每个对象（object）最近的中心点（center_point），并输出<center_point, object>对。reduce()函数计算每个聚类中对象的距离均值，并将这K个均值作为下一轮初始中心点。

(3) 贝叶斯分类

贝叶斯分类是一种利用概率统计知识进行分类的统计学分类方法。该方法包括两个步骤：训练样本和分类。其实现由多个MapReduce作业完成，具体如图2-3所示。其中，训练样本可由三个MapReduce作业实现：第一个作业（ExtractJob）抽取文档特征，该作业只需要Map即可完成；第二个作业（ClassPriorJob）计算类别的先验概率，即统计每个类别中文档的数目，并计算类别概率；第三个作业（ConditionalProbabilityJob）计算单词的条件概率，即统计<label, word>在所有文档中出现的次数并计算单词的条件概率。后两个作业的具体实现类似于WordCount。分类过程由一个作业（PredictJob）完成。该作业的map()函数计算每个待分类文档属于每个类别的概率，reduce()函数找出每个文档概率最高的类别，并输出<docid, label>（编号为docid的文档属于类别label）。

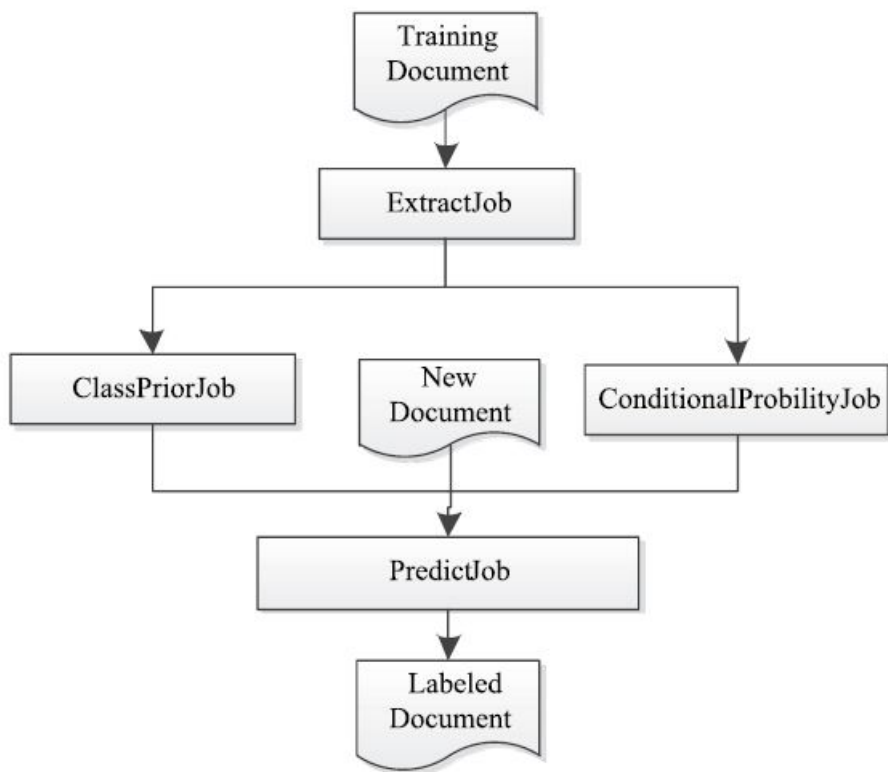


图 2-3 朴素贝叶斯分类算法在MapReduce上实现

前面介绍的是MapReduce可以解决的一些问题。为了便于读者更深刻地理解MapReduce，下面介绍MapReduce不能解决或者难以解决的一些问题。

1) Fibonacci数值计算。Fibonacci数值计算时，下一个结果需要依赖于前面的计算结果，也就是说，无法将该问题划分成若干个互不相干的子问题，因而不能用MapReduce解决。

2) 层次聚类法。层次聚类法是应用最广泛的聚类算法之一。层次聚类法采用迭代控制策略，使聚类逐步优化。它按照一定的相似性（一般是距离）判断标准，合并最相似的部分或者分割最不相似的部分。按采用“自顶向下”和“自底向上”两种方式，可将其分为分解型层次聚类法和聚结型层次聚类法两种。以分解型层次聚类算法为例，其主要思想是，开始时，将每个对象归为一类，然后不断迭代，直到所有对象合并成一个大类（或者达到某个终止条件）；在每轮迭代时，需计算两两对象间的距离，并合并距离最近的两个对象为一类。该算法需要计算两两对象间的距离，也就是说每个对象和其他对象均有关联，因而该问题不能被分解成若干个子问题，进而不能用MapReduce解决。

2.4 Hadoop基本架构

Hadoop由两部分组成，分别是分布式文件系统和分布式计算框架MapReduce。其中，分布式文件系统主要用于大规模数据的分布式存储，而MapReduce则构建在分布式文件系统之上，对存储在分布式文件系统的数据进行分布式计算。本书主要涉及MapReduce，但考虑到它的一些功能跟底层存储机制相关，因而会首先介绍分布式文件系统。

在Hadoop中，MapReduce底层的分布式文件系统是独立模块，用户可按照约定的一套接口实现自己的分布式文件系统，然后经过简单的配置后，存储在该文件系统上的数据便可以被MapReduce处理。Hadoop默认使用的分布式文件系统是HDFS（Hadoop Distributed File System, Hadoop分布式文件系统），它与MapReduce框架紧密结合。本节首先介绍分布式存储系统HDFS的基础架构，然后介绍MapReduce计算框架。

2.4.1 HDFS架构

HDFS是一个具有高度容错性的分布式文件系统，适合部署在廉价的机器上。HDFS能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。

HDFS的架构如图2-4所示，总体上采用了master/slave架构，主要由以下几个组件组成：Client、NameNode、Secondary、NameNode^[1]和DataNode。下面分别对这几个组件进行介绍。

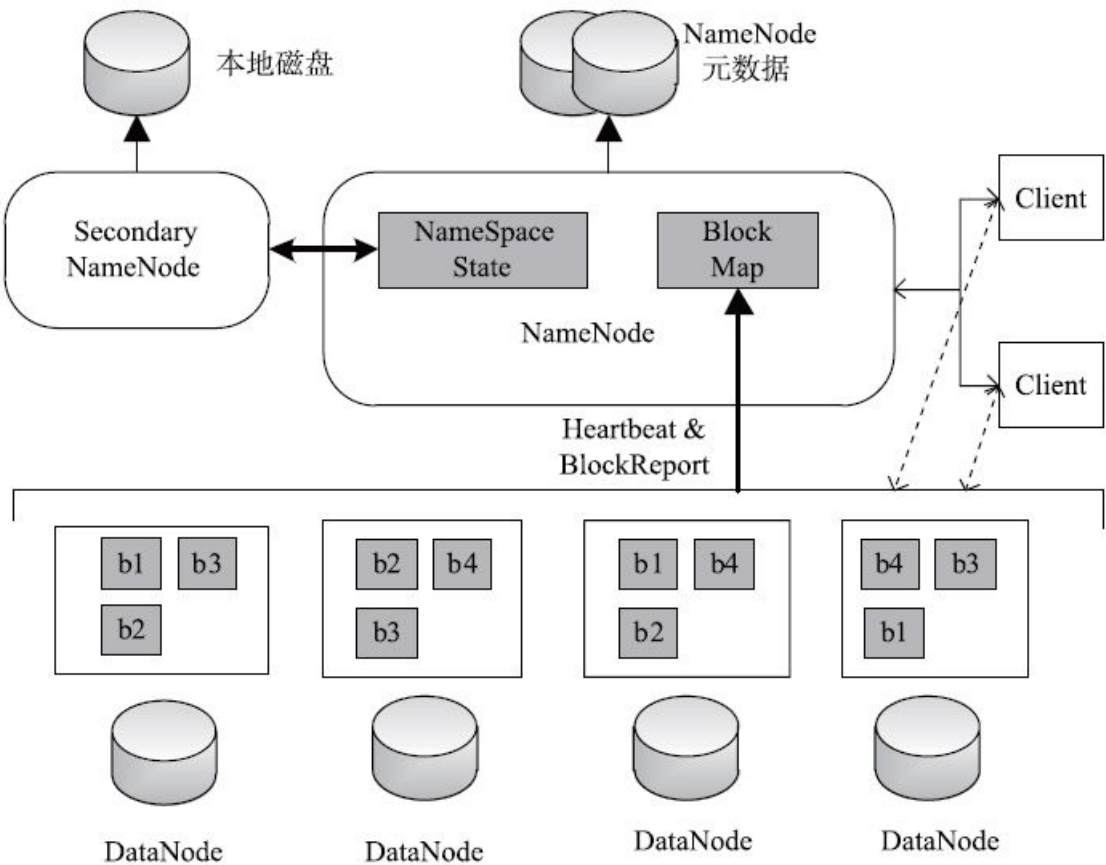


图 2-4 HDFS架构图

(1) Client

Client（代表用户）通过与NameNode和DataNode交互访问HDFS中的文件。Client提供了一个类似POSIX的文件系统接口供用户调用。

（2）NameNode

整个Hadoop集群中只有一个NameNode。它是整个系统的“总管”，负责管理HDFS的目录树和相关的文件元数据信息。这些信息是以“fsimage”（HDFS元数据镜像文件）和“editlog”（HDFS文件改动日志）两个文件^[2]形式存放在本地磁盘，当HDFS重启时重新构造出来的。此外，NameNode还负责监控各个DataNode的健康状态，一旦发现某个DataNode宕掉，则将该DataNode移出HDFS并重新备份其上面的数据。

（3）Secondary NameNode

Secondary NameNode最重要的任务并不是为NameNode元数据进行热备份，而是定期合并fsimage和edits日志，并传输给NameNode。这里需要注意的是，为了减小NameNode压力，NameNode自己并不会合并fsimage和edits，并将文件存储到磁盘上，而是交由Secondary NameNode完成。

（4）DataNode

一般而言，每个Slave节点上安装一个DataNode，它负责实际的数据存储，并将数据信息定期汇报给NameNode。DataNode以固定大小的block为基本单位组织文件内容，默认情况下block大小为64MB。当用户上传一个大的文件到HDFS上时，该文件会被切分成若干个block，分别存储到不同的DataNode；同时，为了保证数据可靠，会将同一个block以流水线方式写到若干个（默认是3，该参数可配置）不同的DataNode上。这种文件切割后存储的过程是对用户透明的。

[1] 在Hadoop 0.21.0版本中，SecondaryNameNode被Checkpoint Node代替。

[2] 在Hadoop 0.21.0版本中，这两个文件被合并成一个。

2.4.2 Hadoop MapReduce架构

同HDFS一样，Hadoop MapReduce也采用了Master/Slave (M/S) 架构，具体如图2-5所示。它主要由以下几个组件组成：Client、JobTracker、TaskTracker和Task。下面分别对这几个组件进行介绍。

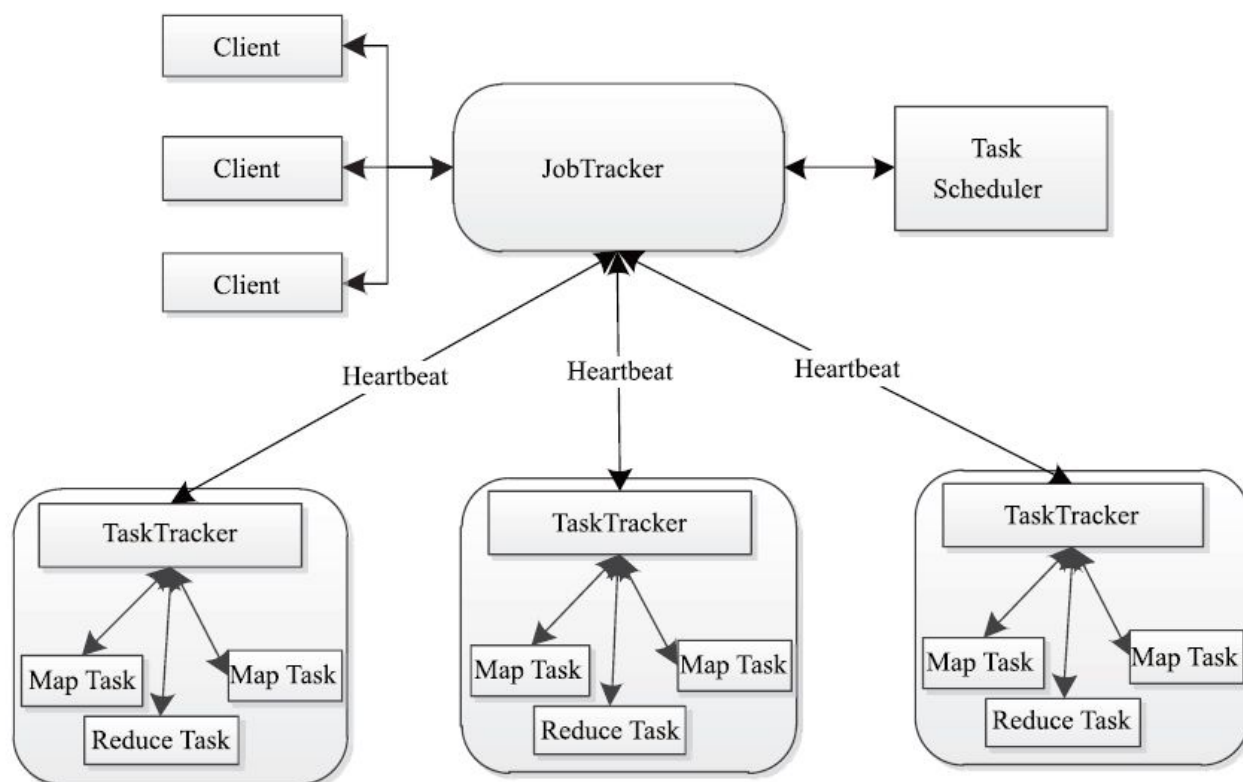


图 2-5 Hadoop MapReduce架构图

(1) Client

用户编写的MapReduce程序通过Client提交到JobTracker端；同时，用户可通过Client提供的一些接口查看作业运行状态。在Hadoop内部用“作业”(Job)表示MapReduce程序。一个MapReduce程序可对应若干个作业，而每个作业会被分解成若干个Map/Reduce任务(Task)。

(2) JobTracker

JobTracker主要负责资源监控和作业调度。JobTracker监控所有TaskTracker与作业的健康状况，一旦发现失败情况后，其会将相应的任务转移到其他节点；同时，JobTracker会跟踪任务的执行进度、资源使用量等信息，并将这些信息告诉任务调度器，而调度器会在资源出现空闲时，选择合适的任务使用这些资源。在Hadoop中，任务调度器是一个可插拔的模块，用户可以根据自己的需要设计相应的调度器。

(3) TaskTracker

TaskTracker会周期性地通过Heartbeat将本节点上资源的使用情况和任务的运行进度汇报给JobTracker，同时接收JobTracker发送过来的命令并执行相应的操作（如启动新任务、杀死任务等）。TaskTracker使用“slot”等量划分本节点上的资源量。“slot”代表计算资源（CPU、内存等）。一个Task获取到一个slot后才有机会运行，而Hadoop调度器的作用就是将各个TaskTracker上的空闲slot分配给Task使用。slot分为Map slot和Reduce slot两种，分别供Map Task和Reduce Task使用。TaskTracker通过slot数目（可配置参数）限定Task的并发度。

(4) Task

Task分为Map Task和Reduce Task两种，均由TaskTracker启动。从上一小节中我们知道，HDFS以固定大小的block为基本单位存储数据，而对于MapReduce而言，其处理单位是split。split与block的对应关系如图2-6所示。split是一个逻辑概念，它只包含一些元数据信息，比如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自己决定。但需要注意的是，split的多少决定了Map Task的数目，因为每个split会交由一个Map Task处理。

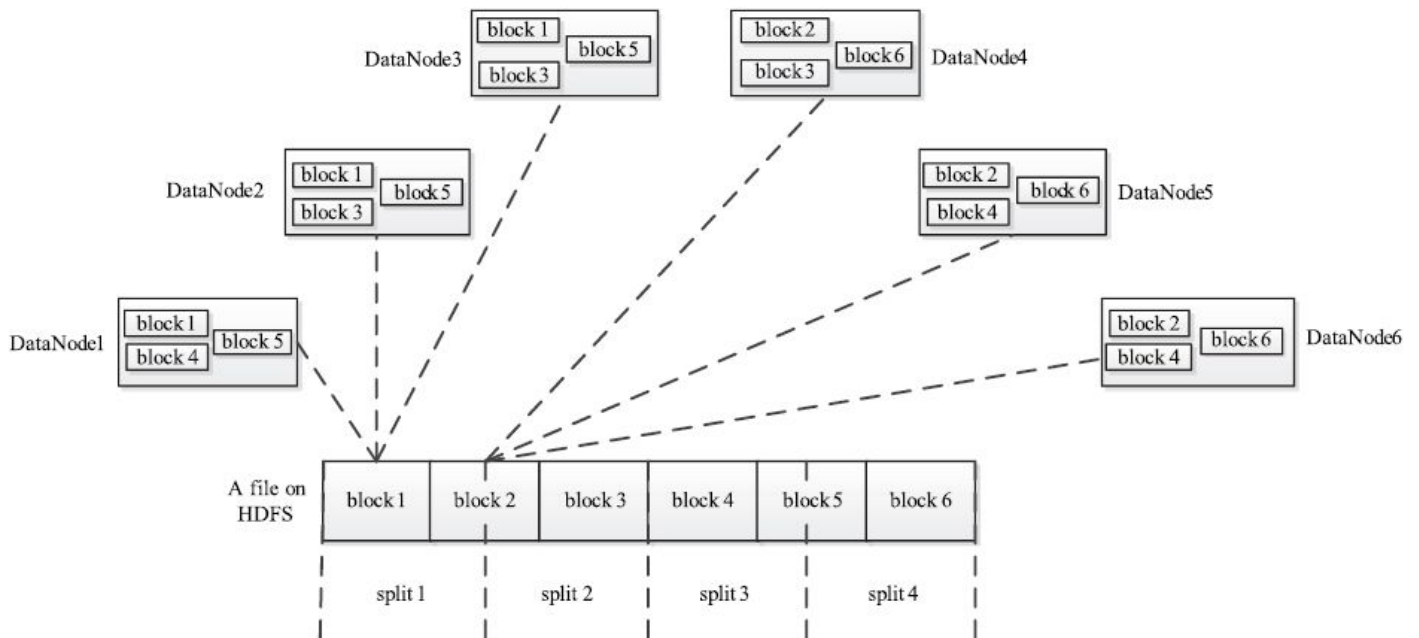


图 2-6 split与block的对应关系

Map Task执行过程如图2-7所示。由该图可知，Map Task先将对应的split迭代解析成一个个key/value对，依次调用用户自定义的map()函数进行处理，最终将临时结果存放到本地磁盘上，其中临时数据被分成若干个partition，每个partition将被一个Reduce Task处理。

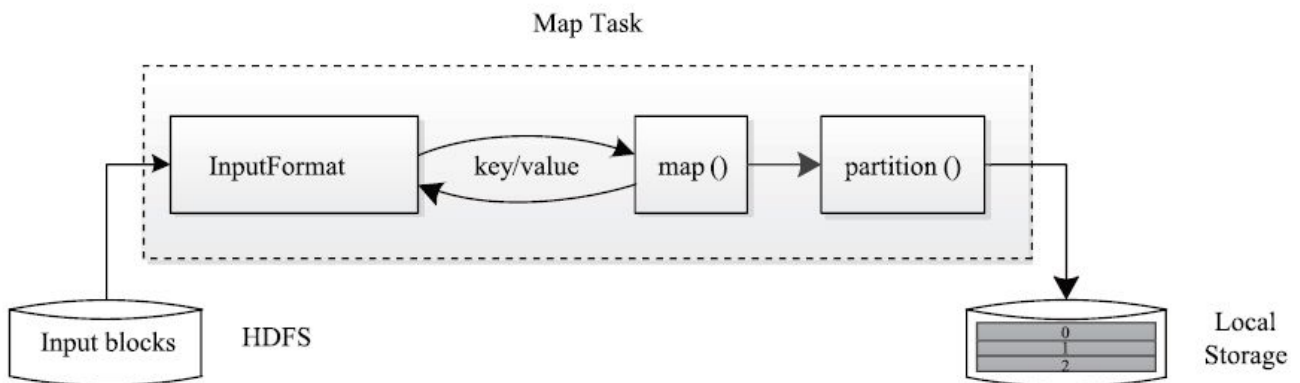


图 2-7 Map Task执行流程

Reduce Task执行过程如图2-8所示。该过程分为三个阶段①从远程节点上读取Map Task中间结果（称为“Shuffle阶段”）；②按照key对key/value对进行排序（称为“Sort阶段”）；③依次读取<key, value list>，调用用户自定义的reduce()函数处理，并将最终结果存到HDFS上（称为“Reduce阶段”）。

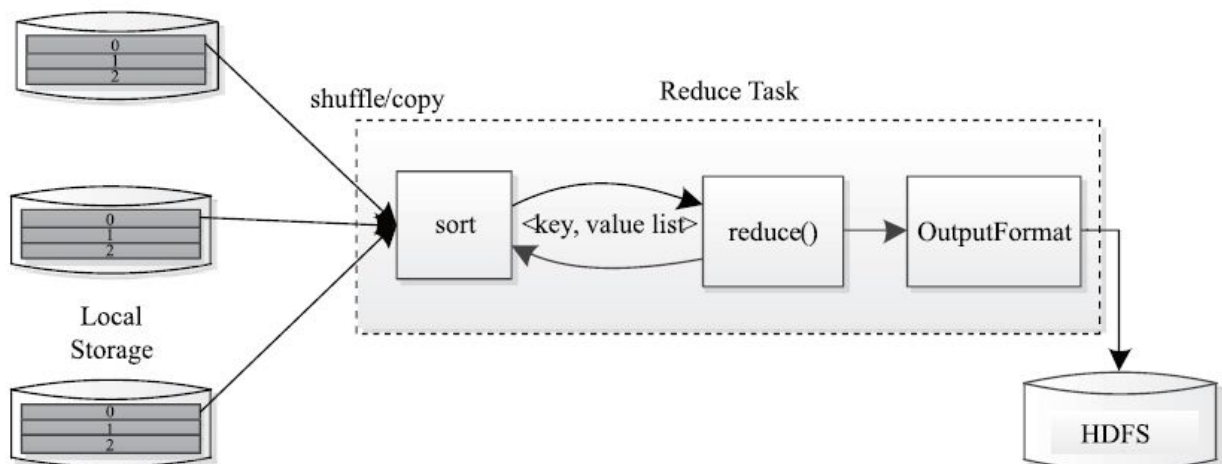


图 2-8 Reduce Task 执行过程

2.5 Hadoop MapReduce作业的生命周期

由于本书以“作业生命周期”为线索对Hadoop MapReduce架构设计和实现原理进行解析，因而在深入剖析各个MapReduce实现细节之前整体了解一个作业的生命周期显得非常重要。为此，本节主要讲解Hadoop MapReduce作业的生命周期，即作业从提交到运行结束经历的整个过程。本节只是概要性地介绍MapReduce作业的生命周期，可看作后续几章的内容导读。作业生命周期中具体各个阶段的深入剖析将在后续的章节中进行。

假设用户编写了一个MapReduce程序，并将其打包成xxx.jar文件，然后使用以下命令提交作业：

```
$HADOOP_HOME/bin/hadoop jar xxx.jar\  
-D mapred.job.name="xxx"\  
-D mapred.map.tasks=3\  
-D mapred.reduce.tasks=2\  
-D input=/test/input\  
-D output=/test/output
```

则该作业的运行过程如图2-9所示。

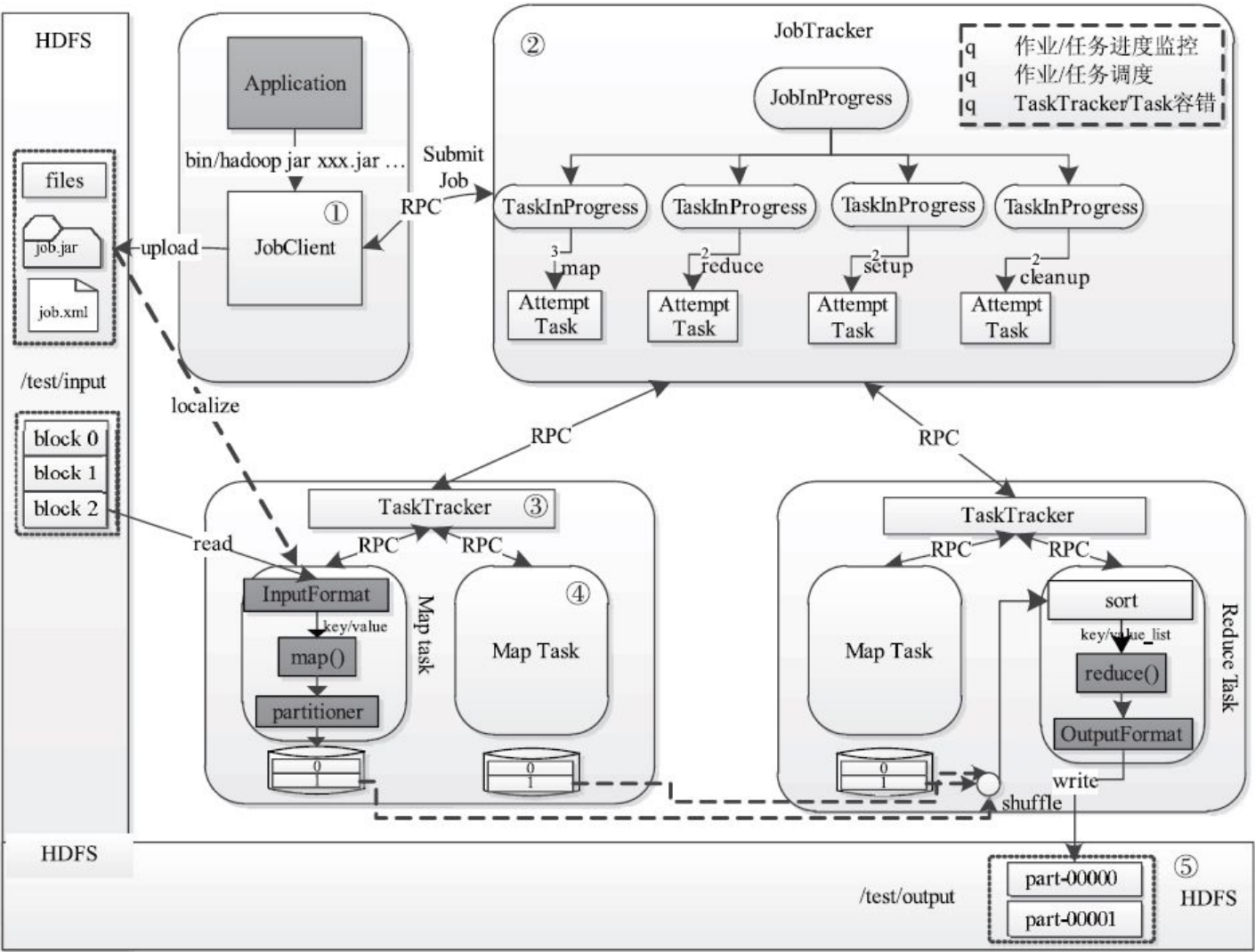


图 2-9 Hadoop MapReduce作业的生命周期

这个过程分为以下5个步骤：

步骤1 作业提交与初始化。用户提交作业后，首先由JobClient实例将作业相关信息，比如将程序jar包、作业配置文件、分片元信息文件等上传到分布式文件系统（一般为HDFS）上，其中，分片元信息文件记录了每个输入分片的逻辑位置信息。然后JobClient通过RPC通知JobTracker。JobTracker收到新作业提交请求后，由作业调度模块对作业进行初始化：为作业创建一个JobInProgress对象以跟踪作业运行状况，而JobInProgress则会为每个Task创建一个TaskInProgress对象以跟踪每个任务的运行状态，TaskInProgress可能需要管理多个“Task运

行尝试”（称为“Task Attempt”）。具体分析见第5章。

步骤2 任务调度与监控。前面提到，任务调度和监控的功能均由JobTracker完成。TaskTracker周期性地通过Heartbeat向JobTracker汇报本节点的资源使用情况，一旦出现空闲资源，JobTracker会按照一定的策略选择一个合适的任务使用该空闲资源，这由任务调度器完成。任务调度器是一个可插拔的独立模块，且为双层架构，即首先选择作业，然后从该作业中选择任务，其中，选择任务时需要重点考虑数据本地性。此外，JobTracker跟踪作业的整个运行过程，并为作业的成功运行提供全方位的保障。首先，当TaskTracker或者Task失败时，转移计算任务；其次，当某个Task执行进度远落后于同一作业的其他Task时，为之启动一个相同Task，并选取计算快的Task结果作为最终结果。具体分析见第6章。

步骤3 任务运行环境准备。运行环境准备包括JVM启动和资源隔离，均由TaskTracker实现。TaskTracker为每个Task启动一个独立的JVM以避免不同Task在运行过程中相互影响；同时，TaskTracker使用了操作系统进程实现资源隔离以防止Task滥用资源。具体分析见第7章。

步骤4 任务执行。TaskTracker为Task准备好运行环境后，便会启动Task。在运行过程中，每个Task的最新进度首先由Task通过RPC汇报给TaskTracker，再由TaskTracker汇报给JobTracker。具体分析见第8章。

步骤5 作业完成。待所有Task执行完毕后，整个作业执行成功。

2.6 小结

Hadoop MapReduce直接诞生于搜索领域，以易于编程、良好的扩展性和高容错性为设计目标。它主要由两部分组成：编程模型和运行时环境。其中，编程模型为用户提供了5个可编程组件，分别是InputFormat、Mapper、Partitioner、Reducer和OutputFormat；运行时环境则将用户的MapReduce程序部署到集群的各个节点上，并通过各种机制保证其成功运行。

Hadoop MapReduce处理的数据一般位于底层分布式文件系统中。该系统往往将用户的文件切分成若干个固定大小的block存储到不同节点上。默认情况下，MapReduce的每个Task处理一个block。MapReduce主要由四个组件构成，分别是Client、JobTracker、TaskTracker和Task，它们共同保障一个作业的成功运行。一个MapReduce作业的运行周期是，先在Client端被提交到JobTracker上，然后由JobTracker将作业分解成若干个Task，并对这些Task进行调度和监控，以保障这些程序运行成功，而TaskTracker则启动JobTracker发来的Task，并向JobTracker汇报这些Task的运行状态和本节点上资源的使用情况。

第二部分 MapReduce编程模型篇

本部分内容

MapReduce编程模型

第3章 MapReduce编程模型

MapReduce应用广泛的原因之一在于它的易用性。它提供了一个因高度抽象化而变得异常简单的编程模型^[1]。在第2章中，我们已经对该编程模型的定义以及应用场景做了简单介绍。在这一章中，我们将从Hadoop MapReduce编程模型实现的角度对其进行深入分析，包括其编程模型的体系结构、设计原理等。

本章不介绍Hadoop MapReduce API的使用方法，而是从实现角度介绍其设计方法。

3.1 MapReduce编程模型概述

在第2章中，我们提到MapReduce是在总结大量应用的共同特点的基础上抽象出来的分布式计算框架，它适用的应用场景往往具有一个共同的特点：任务可被分解成相互独立的子问题。基于该特点，MapReduce编程模型给出了其分布式编程方法，共分5个步骤：

- 1) 迭代（iteration）。遍历输入数据，并将之解析成key/value对。
- 2) 将输入key/value对映射（map）成另外一些key/value对。
- 3) 依据key对中间数据进行分组（grouping）。
- 4) 以组为单位对数据进行归约（reduce）。
- 5) 迭代。将最终产生的key/value对保存到输出文件中。

MapReduce将计算过程分解成以上5个步骤带来的最大好处是组件化与并行化。

为了实现MapReduce编程模型，Hadoop设计了一系列对外编程接口。用户可通过实现这些接口完成应用程序的开发。

3.1.1 MapReduce编程接口体系结构

MapReduce编程模型对外提供的编程接口体系结构如图3-1所示，整个编程模型位于应用程序层和MapReduce执行器之间，可以分为两层。第一层是最基本的Java API，主要有5个可编程组件，分别是InputFormat、Mapper、Partitioner、Reducer和OutputFormat^[2]。Hadoop自带了很多直接可用的InputFormat、Partitioner和OutputFormat，大部分情况下，用户只需编写Mapper和Reducer即可。第二层是工具层，位于基本Java API之上，主要是为了方便用户编写复杂的MapReduce程序和利用其他编程语言增加MapReduce计算平台的兼容性而提出来的。在该层中，主要提供了4个编程工具包。

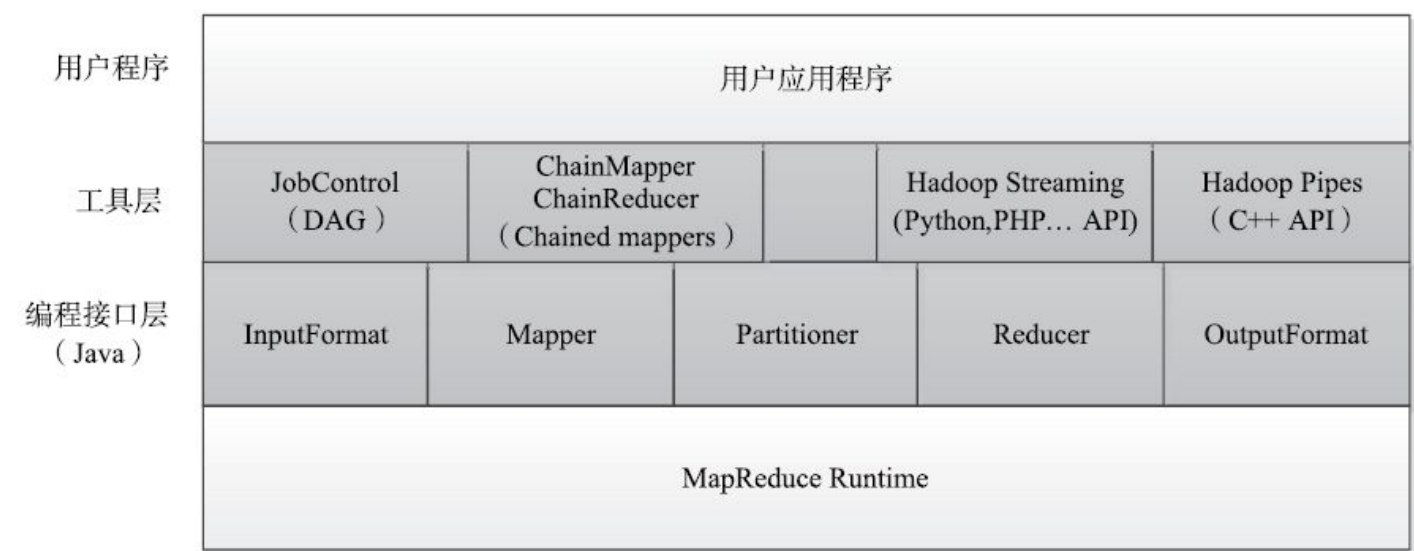


图 3-1 MapReduce编程接口体系结构

❑JobControl: 方便用户编写有依赖关系的作业，这些作业往往构成一个有向图，所以通常称为DAG（Directed Acyclic Graph）作业，如第2章中的朴素贝叶斯分类算法实现便是4个有依赖关系的作业构成的DAG。

❑ChainMapper/ChainReducer: 方便用户编写链式作业，即在Map或者Reduce阶段存在多个Mapper，形式如下：

```
[MAPPER+REDUCER MAPPER*]
```

❑Hadoop Streaming: 方便用户采用非Java语言编写作业，允许用户指定可执行文件或者脚本作为Mapper/Reducer。

❑Hadoop Pipes: 专门为C/C++程序员编写MapReduce程序提供的工具包。

[1] 关于MapReduce编程模型的数学基础，可参考Ralf Lammel的论文“Google’s MapReduce Programming Model—Revisited”。

[2] 还有一个组件是Combiner，它实际是一个Reducer。

3.1.2 新旧MapReduce API比较

从0.20.0版本开始，Hadoop同时提供了新旧两套MapReduce API。新API在旧API基础上进行了封装，使得其在扩展性和易用性方面更好。新旧版MapReduce API的主要区别如下。

(1) 存放位置

旧版API放在org.apache.hadoop.mapred包中，而新版API则放在org.apache.hadoop.mapreduce包及其子包中。

(2) 接口变为抽象类

接口通常作为一种严格的“协议约束”。它只有方法声明而没有方法实现，且要求所有实现类（不包括抽象类）必须实现接口中的每一个方法。接口的最大优点是允许一个类实现多个接口，进而实现类似C++中的“多重继承”。抽象类则是一种较宽松的“约束协议”，它可为某些方法提供默认实现。而继承类则可选择是否重新实现这些方法。正是因为这一点，抽象类在类衍化方面更有优势，也就是说，抽象类具有良好的向后兼容性，当需要为抽象类添加新的方法时，只要新添加的方法提供了默认实现，用户之前的代码就不必修改了。

考虑到抽象类在API衍化方面的优势，新API将InputFormat、OutputFormat、Mapper、Reducer和Partitioner由接口变为抽象类。

(3) 上下文封装

新版API将变量和函数封装成各种上下文（Context）类，使得API具有更好的易用性和扩展性。首先，函数参数列表经封装后变短，使得函数更容易使用；其次，当需要修改或添加某些变量或函数时，只需修改封装后的上下文类即可，用户代码无须修改，这样保证了向后兼容性，具有良好的扩展性。

图3-2展示了新版API中树形的Context类继承关系。这些Context各自封装了一种实体的基本信息及对应的操作（setter和getter函数），如JobContext、TaskAttemptContext分别封装了Job和Task的基本信息，TaskInputOutputContext封装了Task的各种输入输出操作，MapContext和ReduceContext分别封装了Mapper和Reducer对外的公共接口。

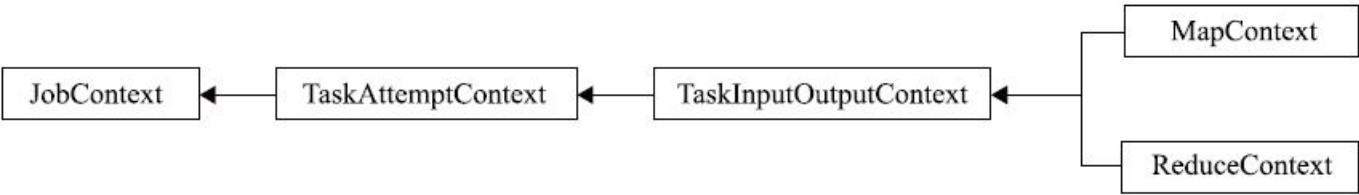


图 3-2 新版API中树形Context类继承关系

除了以上三点不同之外，新旧API在很多其他细节方面也存在小的差别，具体将在接下来的内容中讲解。

由于新版和旧版API在类层次结构、编程接口名称及对应的参数列表等方面存在较大差别，所以两种API不能兼容。但考虑到应用程序的向后兼容性，短时间内不会将旧API从MapReduce中去掉。即使在完全采用新API的0.21.0/0.22.X版本系列中，也仅仅将旧API标注为过期（deprecated），用户仍然可以使用。

本章将对对比介绍两套MapReduce API的设计细节。但考虑到新版API只是在旧版基础上封装而来的，因此，我们将详细分析旧版API的设计思路，而对于新版API，仅是概要介绍它与旧版本的不同之处。

3.2 MapReduce API基本概念

在正式分析新旧API之前，先要介绍几个基本概念。这些概念贯穿于所有API之中，因此，有必要单独讲解。

3.2.1 序列化

序列化是指将结构化对象转为字节流以便于通过网络进行传输或写入持久存储的过程。反序列化指的是将字节流转为结构化对象的过程。在Hadoop MapReduce中，序列化的主要作用有两个：永久存储和进程间通信。

为了能够读取或者存储Java对象，MapReduce编程模型要求用户输入和输出数据中的key和value必须是可序列化的。在Hadoop MapReduce中，使一个Java对象可序列化的方法是让其对应的类实现Writable接口。但对于key而言，由于它是数据排序的关键字，因此还需要提供比较两个key对象的方法。为此，key对应类需实现WritableComparable接口，它的类如图3-3所示 [1]。

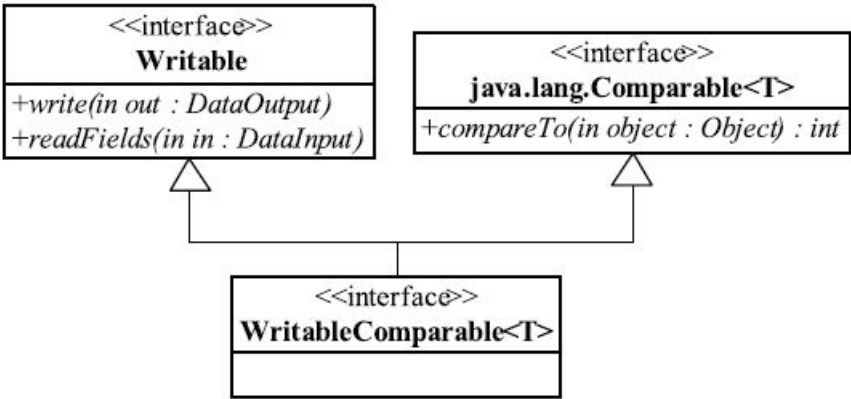


图 3-3 序列化接口WritableComparable的类图

[1] 关于Hadoop序列化的更详细介绍，可参考《Hadoop权威指南》的“第4章Hadoop I/O”。

3.2.2 Reporter参数

Reporter是MapReduce提供给应用程序的工具。如图3-4所示，应用程序可使用Reporter中的方法报告完成进度（progress）、设定状态消息（setStatus）以及更新计数器（incrCounter）。

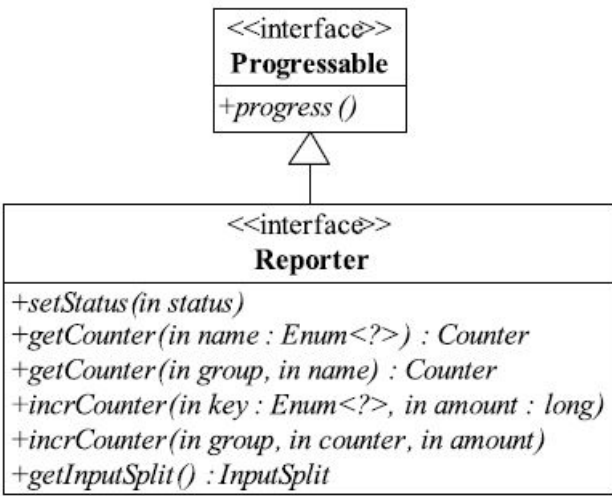


图 3-4 Reporter的类图

Reporter是一个基础参数。MapReduce对外提供的大部分组件，包括InputFormat、Mapper和Reducer等，均在其主要方法中添加了该参数，具体可参考3.3节。

3.2.3 回调机制

回调机制是一种常见的设计模式。它将 workflow 内的某个功能按照约定的接口暴露给外部使用者，为外部使用者提供数据，或要求外部使用者提供数据。

Hadoop MapReduce 对外提供的 5 个组件（InputFormat、Mapper、Partitioner、Reducer 和 OutputFormat）实际上全部属于回调接口。当用户按照约定实现这几个接口后，MapReduce 运行时环境会自动调用它们。

如图 3-5 所示，MapReduce 给用户暴露了接口 Mapper，当用户按照自己的应用程序逻辑实现自己的 MyMapper 后，Hadoop MapReduce 运行时环境会将输入数据解析成 key/value 对，并调用 map() 函数迭代处理。

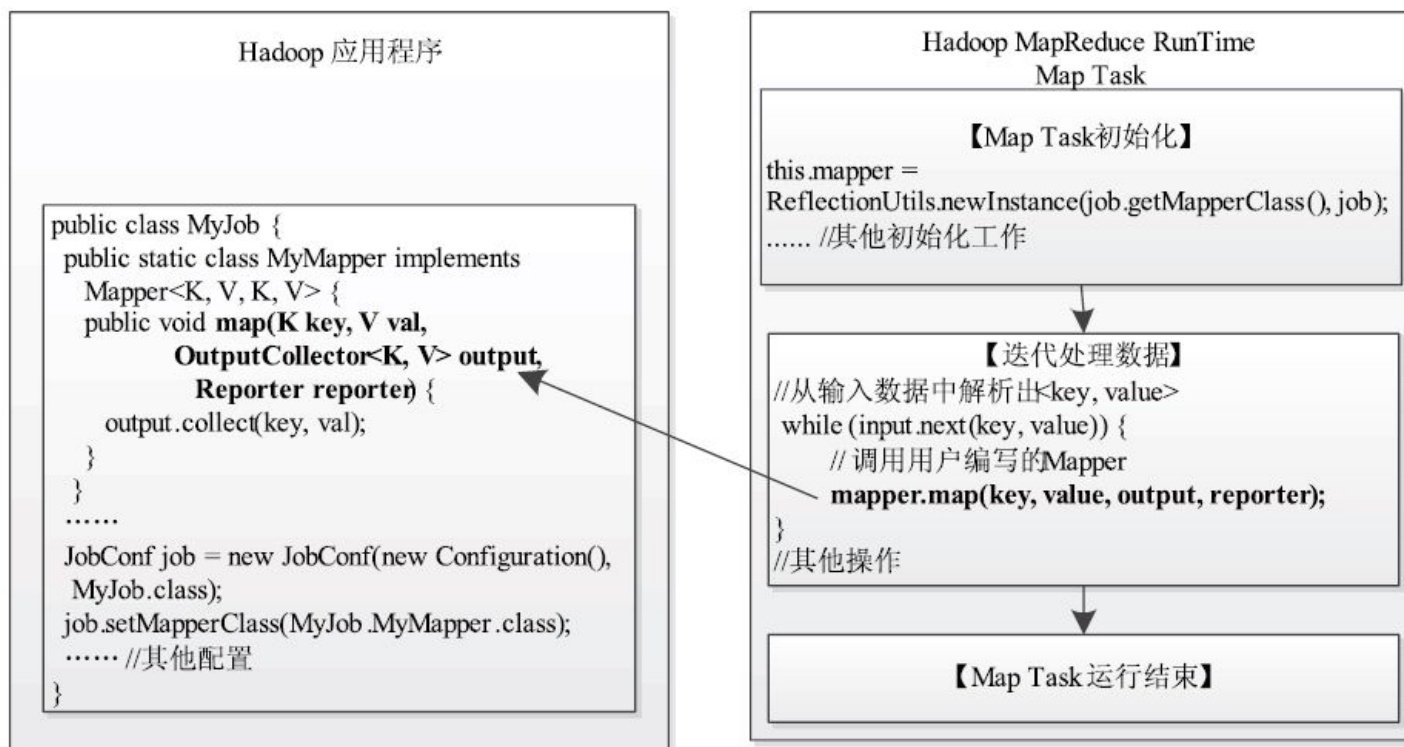


图 3-5 MapReduce回调机制实例

3.3 Java API解析

Hadoop的主要编程语言是Java，因而Java API是最基本的对外编程接口。当前各个版本的Hadoop均同时存在新旧两种API。本节将对讲解这两种API的设计思路，主要内容包括使用实例、接口设计、在MapReduce运行时环境中的调用时机等。

3.3.1 作业配置与提交

1.Hadoop配置文件介绍

在Hadoop中，Common、HDFS和MapReduce各有对应的配置文件，用于保存对应模块中可配置的参数。这些配置文件均为XML格式且由两部分构成：系统默认配置文件和管理员自定义配置文件。其中，系统默认配置文件分别是core-default.xml、hdfs-default.xml和mapred-default.xml，它们包含了所有可配置属性的默认值。而管理员自定义配置文件分别是core-site.xml、hdfs-site.xml和mapred-site.xml。它们由管理员设置，主要用于定义一些新的配置属性或者覆盖系统默认配置文件中的默认值。通常这些配置一旦确定，便不能被修改（如果想修改，需重新启动Hadoop）。需要注意的是，core-default.xml和core-site.xml属于公共基础库的配置文件，默认情况下，Hadoop总会优先加载它们。

在Hadoop中，每个配置属性主要包括三个配置参数：name、value和description，分别表示属性名、属性值和属性描述。其中，属性描述仅仅用来帮助用户理解属性的含义，Hadoop内部并不会使用它的值。此外，Hadoop为配置文件添加了两个新的特性：final参数和变量扩展。

□final参数：如果管理员不想让用户程序修改某些属性的属性值，可将该属性的final参数置为true，比如：

```
<property>
<name>mapred.map.tasks.speculative.execution</name>
<value>true</value>
<final>true</final>
</property>
```

管理员一般在XXX-site.xml配置文件中为某些属性添加final参数，以防止用户在应用程序中修改这些属性的属性值。

□变量扩展：当读取配置文件时，如果某个属性存在对其他属性的引用，则Hadoop首先会查找引用的属性是否为下列两种属性之一。如果是，则进行扩展。

○其他已经定义的属性。

○Java中System.getProperties()函数可获取属性。

比如，如果一个配置文件中包含以下配置参数：

```
<property>
<name>hadoop.tmp.dir</name>
<value>/tmp/hadoop-${user.name}</value>
</property>
<property>
<name>mapred.temp.dir</name>
<value>${hadoop.tmp.dir}/mapred/temp</value>
</property>
```

则当用户想要获取属性mapred.temp.dir的值时，Hadoop会将hadoop.tmp.dir解析成该配置文件中另外一个属性的值，而user.name则被替换成系统属性user.name的值。

2.MapReduce作业配置与提交

在MapReduce中，每个作业由两部分组成：应用程序和作业配置。其中，作业配置内容包括环境配置和用户自定义配置两部分。环境配置由Hadoop自动添加，主要由mapred-default.xml和mapred-site.xml两个文件中的配置选项组合而成；用户自定义配置则由用户自己根据作业特点个性化定制而成，比如用户可设置作业名称，以及Mapper/Reducer、Reduce Task个数等。在新旧两套API中，作业配置接口发生了变化，首先通过一个例子感受一下使用上的不同。

旧API作业配置实例：

```
JobConf job=new JobConf (new Configuration(), MyJob.class);
job.setJobName ("myjob");
job.setMapperClass (MyJob.MyMapper.class);
job.setReducerClass (MyJob.MyReducer.class);
JobClient.runJob (job);
```

新API作业配置实例：

```
Configuration conf=new Configuration();
Job job=new Job (conf, "myjob");
job.setJarByClass (MyJob.class);
job.setMapperClass (MyJob.MyMapper.class);
job.setReducerClass (MyJob.MyReducer.class);
System.exit (job.waitForCompletion (true) ? 0: 1);
```

从以上两个实例可以看出，新版API用Job类代替了JobConf和JobClient两个类，这样，仅使用一个类的同时可完成作业配置和作业提交相关功能，进一步简化了作业编写方式。我们将在第5章介绍作业提交的相关细节，本小节重点从设计角度分析新旧两套API中作业配置的相关实现细节。

3.旧API中的作业配置

MapReduce配置模块代码结构如图3-6所示。其中，org.apache.hadoop.conf中的Configuration类是配置模块最底层的类。从图3-6中可以看出，该类支持以下两种基本操作。

- ❑序列化：序列化是将结构化数据转换成字节流，以便于传输或存储。Java实现了自己的一套序列化框架。凡是需要支持序列化的类，均需要实现Writable接口。
- ❑迭代：为了方便遍历所有属性，它实现了Java开发包中的Iterator接口。

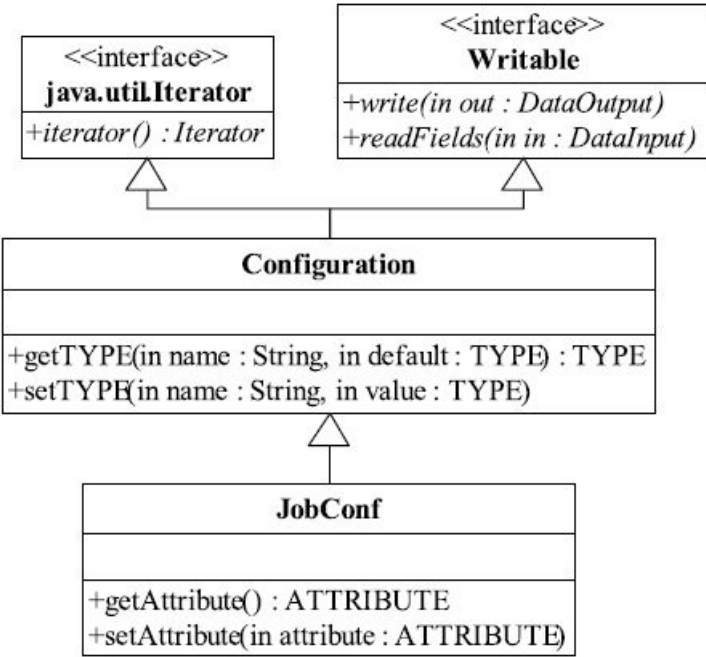


图 3-6 旧MapReduce API中作业配置类图

Configuration类总会依次加载core-default.xml和core-site.xml两个基础配置文件，相关代码如下：

```
addDefaultResource ("core-default.xml");
addDefaultResource ("core-site.xml");
```

addDefaultResource函数的参数为XML文件名，它能够将XML文件中的name/value加载到内存中。当连续调用多次该函数时，对于同一个配置选项，其后面的值会覆盖前面的值。

Configuration类中有大量针对常见数据类型的getter/setter函数，用于获取或者设置某种数据类型属性的属性值。比如，对于float类型，提供了这样一对函数：

```
float getFloat (String name, float defaultValue)
void setFloat (String name, float value)
```

除了大量getter/setter函数外，Configuration类中还有一个非常重要的函数：

```
void writeXml (OutputStream out)
```

该函数能够将当前Configuration对象中所有属性及属性值保存到一个XML文件中，以便于在节点之间传输。这点在以后的几节中会提到。

JobConf描述了一个MapReduce作业运行时需要的所有信息，而MapReduce运行时环境正是根据JobConf提供的信息运行作业的。

JobConf继承了Configuration类，并添加了一些设置/获取作业属性的setter/getter函数，以方便用户编写MapReduce程序，如设置/获取Reduce Task个数的函数为：

```
public int getNumReduceTasks () {return getInt ("mapred.reduce.tasks", 1); }
public void setNumReduceTasks (int n) {setInt ("mapred.reduce.tasks", n); }
```

JobConf中添加的函数均是对Configuration类中函数的再次封装。由于它在这些函数名中融入了作业属性的名字，因而更易于使用。

默认情况下，JobConf会自动加载配置文件mapred-default.xml和mapred-site.xml，相关代码如下：

```
static{
Configuration.addDefaultResource ("mapred-default.xml");
Configuration.addDefaultResource ("mapred-site.xml");
}
```

4.新API中的作业配置

前面提到，与新API中的作业配置相关的类是Job。该类同时具有作业配置和作业提交的功能，其中，作业提交将在第5章中介绍，这里只关注作业配置部分。作业配置部分的类图如图3-7所示。Job类继承了一个新类JobContext，而Context自身则包含一个JobConf类型的成员。注意，JobContext类仅提供了一些getter方法，而Job类中则提供了一些setter方法。

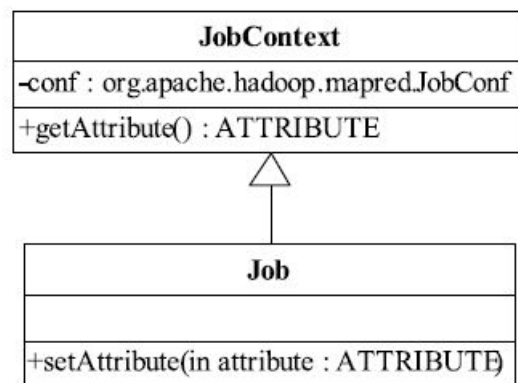


图 3-7 新MapReduce API中作业配置类图

3.3.2 InputFormat接口的设计与实现

InputFormat主要用于描述输入数据的格式，它提供以下两个功能。

□数据切分：按照某个策略将输入数据切分成若干个split，以便确定Map Task个数以及对应的split。

□为Mapper提供输入数据：给定某个split，能将其解析成一个个key/value对。

本小节将介绍Hadoop如何设计InputFormat接口，以及提供了哪些常用的InputFormat实现。

1.旧版API的InputFormat解析

如图3-8所示，在旧版API中，InputFormat是一个接口，它包含两种方法：

```
InputSplit[] getSplits (JobConf job, int numSplits) throws IOException;
RecordReader<K, V> getRecordReader (InputSplit split,
JobConf job,
Reporter reporter) throws IOException;
```

getSplits方法主要完成数据切分的功能，它会尝试着将输入数据切分成numSplits个InputSplit。InputSplit有以下两个特点。

□逻辑分片：它只是在逻辑上对输入数据进行分片，并不会在磁盘上将其切分成分片进行存储。InputSplit只记录了分片的元数据信息，比如起始位置、长度以及所在的节点列表等。

□可序列化：在Hadoop中，对象序列化主要有两个作用：进程间通信和永久存储。此处，InputSplit支持序列化操作主要是为了进程间通信。作业被提交到JobTracker之前，Client会调用作业InputFormat中的getSplits函数，并将得到的InputSplit序列化到文件中。这样，当作业提交到JobTracker端对作业初始化时，可直接读取该文件，解析出所有InputSplit，并创建对应的Map Task。

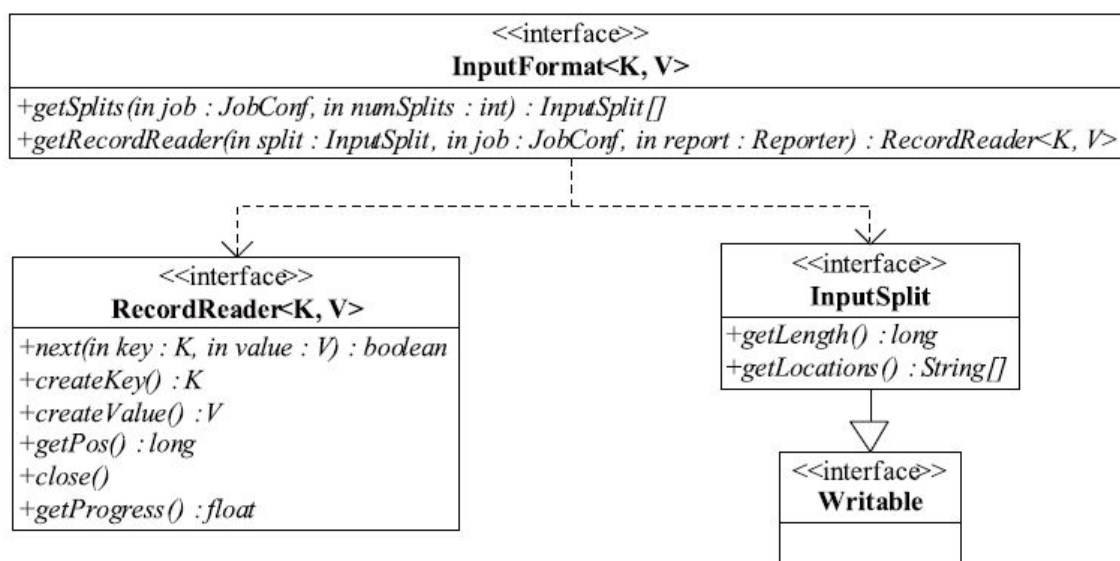


图 3-8 旧API中InputFormat类图

getRecordReader方法返回一个RecordReader对象，该对象可将输入的InputSplit解析成若干个key/value对。MapReduce框架在Map Task执行过程中，会不断调用RecordReader对象中的方法，迭代获取key/value对并交给map()函数处理，主要代码（经过简化）如下：

```
//调用InputSplit的getRecordReader方法获取RecordReader<K1, V1>input
.....
K1 key=input.createKey();
V1 value=input.createValue();
while (input.next(key, value)) {
//调用用户编写的map()函数
}
input.close();
```

前面分析了InputFormat接口的定义，接下来介绍系统自带的各种InputFormat实现。为了方便用户编写MapReduce程序，Hadoop自带了

一些针对数据库和文件的InputFormat实现，具体如图3-9所示。通常而言，用户需要处理的数据均以文件形式存储到HDFS上，所以我们重点针对文件的InputFormat实现进行讨论。

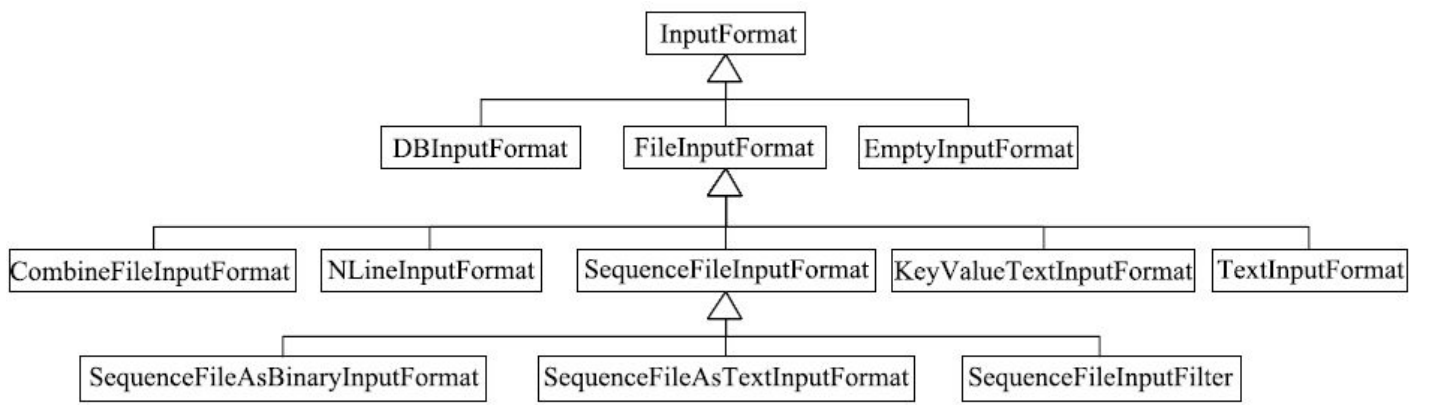


图 3-9 Hadoop MapReduce自带InputFormat实现的类层次图

如图3-9所示，所有基于文件的InputFormat实现的基类是FileInputFormat，并由此派生出针对文本文件格式的TextInputFormat、KeyValueTextInputFormat和NLineInputFormat，针对二进制文件格式的SequenceFileInputFormat等。整个基于文件的InputFormat体系的设计思路是，由公共基类FileInputFormat采用统一的方法对各种输入文件进行切分，比如按照某个固定大小等分，而由各个派生InputFormat自己提供机制将进一步解析InputSplit。对应到具体的实现是，基类FileInputFormat提供getSplits实现，而派生类提供getRecordReader实现。

为了帮助读者深入理解这些InputFormat的实现原理，我们选取TextInputFormat与SequenceFileInputFormat进行重点介绍。

我们首先介绍基类FileInputFormat的实现。它最重要的功能是为各种InputFormat提供统一的getSplits函数。该函数实现中最核心的两个算法是文件切分算法和host选择算法。（1）文件切分算法文件切分算法主要用于确定InputSplit的个数以及每个InputSplit对应的数据段。FileInputFormat以文件为单位切分生成InputSplit。对于每个文件，由以下三个属性值确定其对应的InputSplit的个数。

□ goalSize: 它是根据用户期望的InputSplit数目计算出来的，即totalSize/numSplits。其中，totalSize为文件总大小；numSplits为用户设定的Map Task个数，默认情况下是1。

□ minSize: InputSplit的最小值，由配置参数mapred.min.split.size确定，默认是1。

□ blockSize: 文件在HDFS中存储的block大小，不同文件可能不同，默认是64 MB。这三个参数共同决定InputSplit的最终大小，计算方法如下：

splitSize=max{minSize, min{goalSize, blockSize}}

一旦确定splitSize值后，FileInputFormat将文件依次切成大小为splitSize的InputSplit，最后剩下不足splitSize的数据块单独成为一个InputSplit。

【实例】输入目录下有三个文件file1、file2和file3，大小依次为1 MB，32 MB和250 MB。若blockSize采用默认值64 MB，则不同minSize和goalSize下，file3切分结果如表3-1所示（三种情况下，file1与file2切分结果相同，均为1个InputSplit）。

表 3-1 minSize、goalSize、splitSize 与 InputSplit 对应关系

minSize	goalSize	splitSize	file3 对应的 InputSplit 数目	输入目录对应的 InputSplit 总数
1 MB	totalSize (numSplits=1)	64 MB	4	6
32 MB	totalSize/5	50 MB	5	7
128 MB	totalSize/2	128 MB	2	4

结合表和公式可以知道，如果想让InputSplit尺寸大于block尺寸，则直接增大配置参数mapred.min.split.size即可。

(2) host选择算法

待InputSplit切分方案确定后，下一步要确定每个InputSplit的元数据信息。这通常由四部分组成：<file, start, length, hosts>，分别表示InputSplit所在的文件、起始位置、长度以及所在的host（节点）列表。其中，前三项很容易确定，难点在于host列表的选择方法。

InputSplit的host列表选择策略直接影响到运行过程中的任务本地性。第2章介绍Hadoop架构时，我们提到HDFS上的文件是以block为单位组织的，一个大文件对应的block可能遍布整个Hadoop集群，而InputSplit的划分算法可能导致一个InputSplit对应多个block，这些block可能位于不同节点上，这使得Hadoop不可能实现完全的数据本地性。为此，Hadoop将数据本地性按照代价划分成三个等级：node locality、rack locality和data center locality（Hadoop还未实现该locality级别）。在进行任务调度时，会依次考虑这3个节点的locality，即优先让空闲资源处理本节点上的数据，如果节点上没有可处理的数据，则处理同一个机架上的数据，最差情况是处理其他机架上的数据（但是必须位于同一个数据中心）。

虽然InputSplit对应的block可能位于多个节点上，但考虑到任务调度的效率，通常不会把所有节点加到InputSplit的host列表中，而是选择包含（该InputSplit）数据总量最大的前几个节点（Hadoop限制最多选择10个，多余的会过滤掉），以作为任务调度时判断任务是否具有本地性的主要凭证。为此，FileInputFormat设计了一个简单有效的启发式算法：首先按照rack包含的数据量对rack进行排序，然后在rack内部按照每个node包含的数据量对node排序，最后取前N个node的host作为InputSplit的host列表，这里的N为block副本数。这样，当任务调度器调度Task时，只要将Task调度给位于host列表的节点，就认为该Task满足本地性。

【实例】某个Hadoop集群的网络拓扑结构如图3-10所示，HDFS中block副本数为3，某个InputSplit包含3个block，大小依次是100、150和75，很容易计算，4个rack包含的（该InputSplit的）数据量分别是175、250、150和75。rack2中的node3和node4，rack1中的node1将被添加到该InputSplit的host列表中。

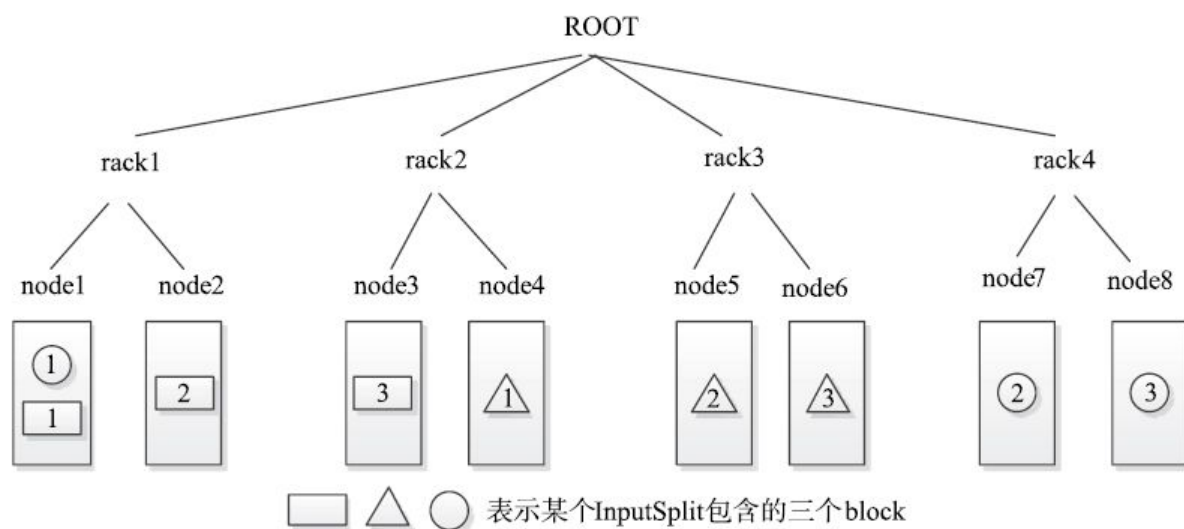


图 3-10 一个Hadoop集群的网络拓扑结构图

从以上host选择算法可知，当InputSplit尺寸大于block尺寸时，Map Task并不能实现完全数据本地性，也就是说，总有一部分数据需要从远程节点上读取，因而可以得出以下结论：

当使用基于FileInputFormat实现InputFormat时，为了提高Map Task的数据本地性，应尽量使InputSplit大小与block大小相同。

分析完FileInputFormat实现方法，接下来分析派生类TextInputFormat与Sequence-FileInputFormat的实现。

前面提到，由派生类实现getRecordReader函数，该函数返回一个RecordReader对象。它实现了类似于迭代器的功能，将某个InputSplit解析成一个个key/value对。在具体实现时，RecordReader应考虑以下两点。

□ 定位记录边界：为了能够识别一条完整的记录，记录之间应该添加一些同步标识。对于TextInputFormat，每两条记录之间存在换行符；对于SequenceFileInputFormat，每隔若干条记录会添加固定长度的同步字符串。通过换行符或者同步字符串，它们很容易定位到一个完整记录的起始位置。另外，由于FileInputFormat仅仅按照数据量多少对文件进行切分，因而InputSplit的第一条记录和最后一条记录可能会被

从中间切开。为了解决这种记录跨越InputSplit的读取问题，RecordReader规定每个InputSplit的第一条不完整记录划给前一个InputSplit处理。

□解析key/value: 定位到一条新的记录后，需将该记录分解成key和value两部分。对于TextInputFormat，每一行的内容即为value，而该行在整个文件中的偏移量为key。对于SequenceFileInputFormat，每条记录的格式为：

```
[record length][key length][key][value]
```

其中，前两个字段分别是整条记录的长度和key的长度，均为4字节，后两个字段分别是key和value的内容。知道每条记录的格式后，很容易解析出key和value。

2.新版API的InputFormat解析

新版API的InputFormat类图如图3-11所示。新API与旧API比较，在形式上发生了较大变化，但仔细分析，发现仅仅是对之前的一些类进行了封装。正如3.1.2节介绍的那样，通过封装，使接口的易用性和扩展性得以增强。

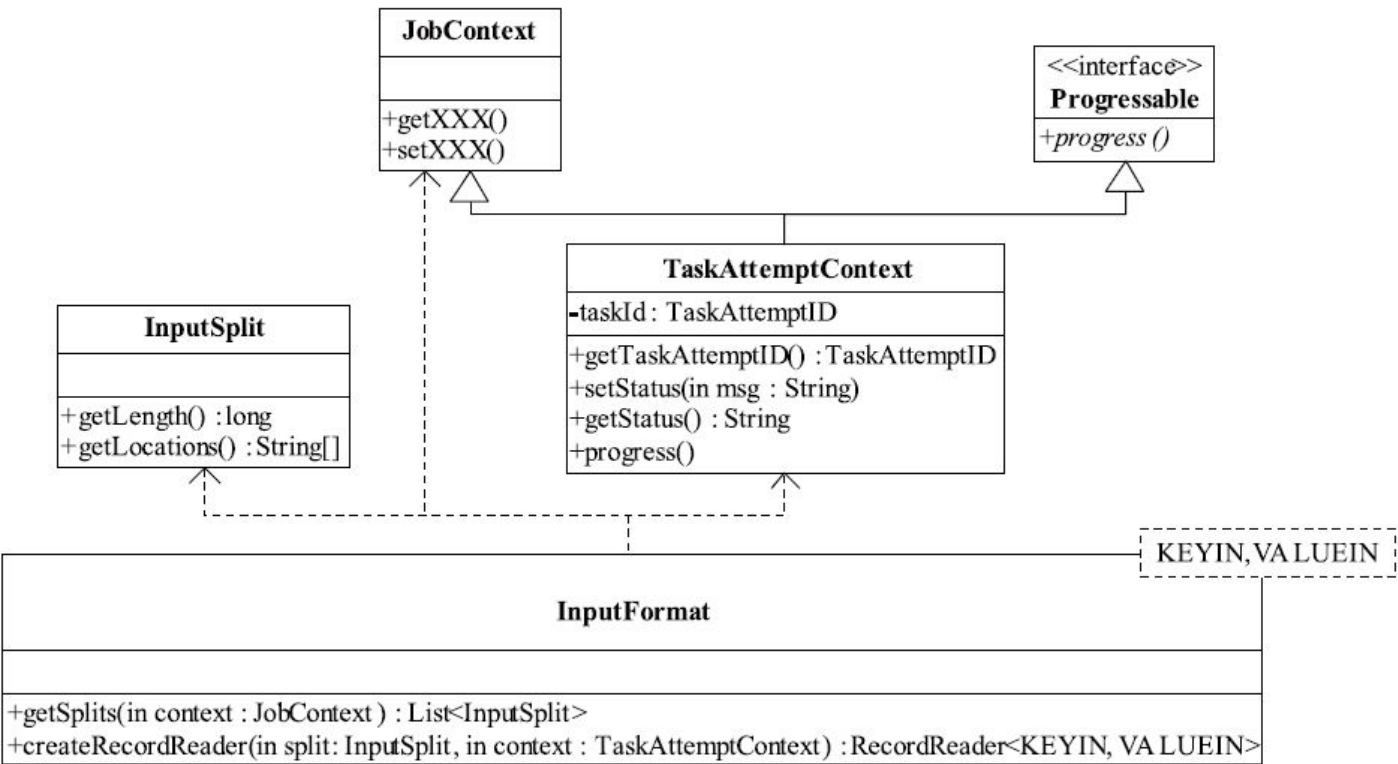


图 3-11 新API中InputFormat类图

此外，对于基类FileInputFormat，新版API中有一个值得注意的改动：InputSplit划分算法不再考虑用户设定的Map Task个数，而用mapred.max.split.size（记为maxSize）代替，即InputSplit大小的计算公式变为：

splitSize=max{minSize, min{maxSize, blockSize}}

3.3.3 OutputFormat接口的设计与实现

OutputFormat主要用于描述输出数据的格式，它能够将用户提供的key/value对写入特定格式的文件中。本小节将介绍Hadoop如何设计OutputFormat接口，以及一些常用的OutputFormat实现。

1.旧版API的OutputFormat解析

如图3-12所示，在旧版API中，OutputFormat是一个接口，它包含两个方法：

```
RecordWriter<K, V>getRecordWriter (FileSystem ignored, JobConf job,
String name, Progressable progress)
throws IOException;
void checkOutputSpecs (FileSystem ignored, JobConf job) throws IOException;
```

checkOutputSpecs方法一般在用户作业被提交到JobTracker之前，由JobClient自动调用，以检查输出目录是否合法。

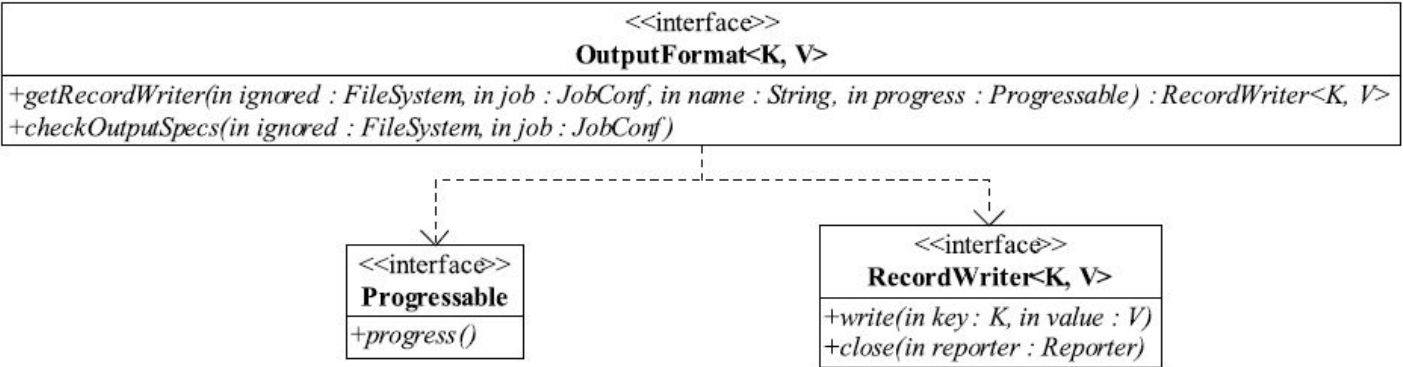


图 3-12 旧版API的OutputFormat类图

getRecordWriter方法返回一个RecordWriter类对象。该类中的方法write接收一个key/value对，并将之写入文件。在Task执行过程中，MapReduce框架会将map()或者reduce()函数产生的结果传入write方法，主要代码（经过简化）如下。

假设用户编写的map()函数如下：

```
public void map (Text key, Text value,
OutputCollector<Text, Text>output,
Reporter reporter) throws IOException{
//根据当前key/value产生新的输出<newKey, newValue>，并输出
.....
output.collect (newKey, newValue);
}
```

则函数output.collect (newKey, newValue) 内部执行代码如下：

```
RecordWriter<K, V>out=job.getOutputFormat().getRecordWriter (.....);
out.write (newKey, newValue);
```

Hadoop自带了很多OutputFormat实现，它们与InputFormat实现相对应，具体如图3-13所示。所有基于文件的OutputFormat实现的基类为FileOutputFormat，并由此派生出一些基于文本文件格式、二进制文件格式的或者多输出的实现。

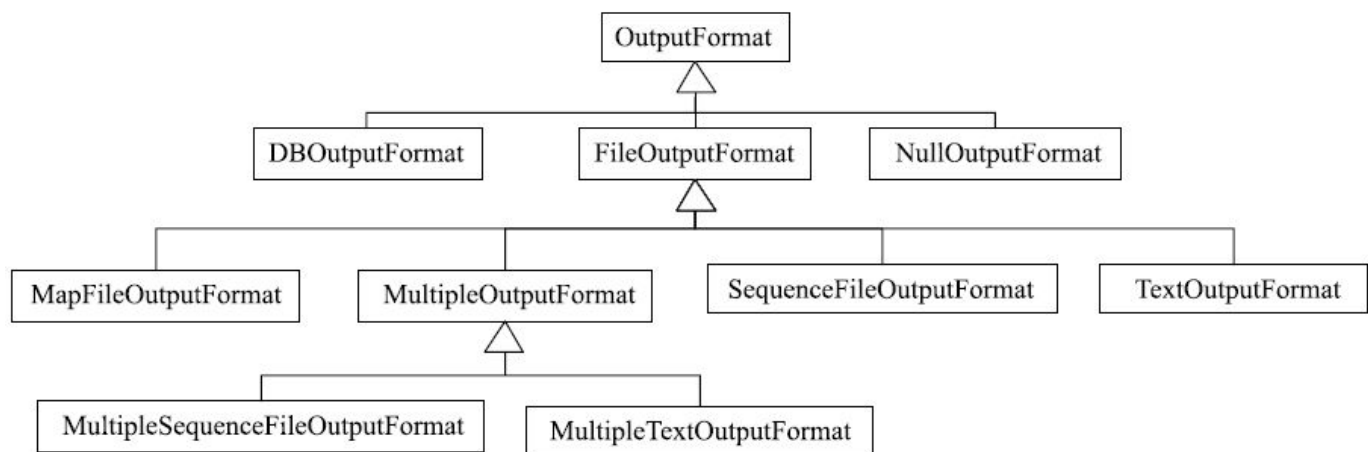


图 3-13 Hadoop MapReduce自带OutputFormat实现的类层次图

为了深入分析OutputFormat的实现方法，我们选取比较有代表性的FileOutputFormat类进行分析。同介绍InputFormat实现的思路一样，我们先介绍基类FileOutputFormat，再介绍其派生类TextOutputFormat。

基类FileOutputFormat需要提供所有基于文件的OutputFormat实现的公共功能，总结起来，主要有以下两个：

（1）实现checkOutputSpecs接口

该接口在作业运行之前被调用，默认功能是检查用户配置的输出目录是否存在，如果存在则抛出异常，以防止之前的数据被覆盖。

（2）处理side-effect file

任务的side-effect file并不是任务的最终输出文件，而是具有特殊用途的任务专属文件。它的典型应用是执行推测式任务。在Hadoop中，因为硬件老化、网络故障等原因，同一个作业的某些任务执行速度可能明显慢于其他任务，这种任务会拖慢整个作业的执行速度。为了对这种“慢任务”进行优化，Hadoop会为之在另外一个节点上启动一个相同的任务，该任务便被称为推测式任务，最先完成任务的计算结果便是这块数据对应的处理结果。为防止这两个任务同时往一个输出文件中写入数据时发生写冲突，FileOutputFormat会为每个Task的数据创建一个side-effect file，并将产生的数据临时写入该文件，待Task完成后，再移动到最终输出目录中。这些文件的相关操作，比如创建、删除、移动等，均由OutputCommitter完成。它是一个接口，Hadoop提供了默认实现FileOutputCommitter，用户也可以根据自己的需求编写OutputCommitter实现，并通过参数{mapred.output.committer.class}指定。OutputCommitter接口定义以及FileOutputCommitter对应的实现如表3-2所示。

表 3-2 OutputCommitter 接口定义以及 FileOutputCommitter 对应的实现

方法	何时被调用	FileOutputCommitter 实现
setupJob	作业初始化	创建临时目录 \${mapred.out.dir}/_temporary
commitJob	作业成功运行完成	删除临时目录，并在 \${mapred.out.dir} 目录下创建空文件 _SUCCESS
abortJob	作业运行失败	删除临时目录
setupTask	任务初始化	不进行任何操作。原本是需要临时目录下创建 side-effect file 的，但它是用时创建的（create on demand）
needsTaskCommit	判断是否需要提交结果	只要存在 side-effect file，就返回 true
commitTask	任务成功运行完成	提交结果，即将 side-effect file 移动到 \${mapred.out.dir} 目录下
abortTask	任务运行失败	删除任务的 side-effect file

注意 默认情况下，当作业成功运行完成后，会在最终结果目录\${mapred.out.dir}下生成空文件_SUCCESS。该文件主要为高层应用提供作业运行完成的标识，比如，Oozie^[1]需要通过检测结果目录下是否存在该文件判断作业是否运行完成。

2.新版API的OutputFormat解析

如图3-14所示，除了接口变为抽象类外，新API中的OutputFormat增加了一个新的方法：getOutputCommitter，以允许用户自己定制合适的OutputCommitter实现。

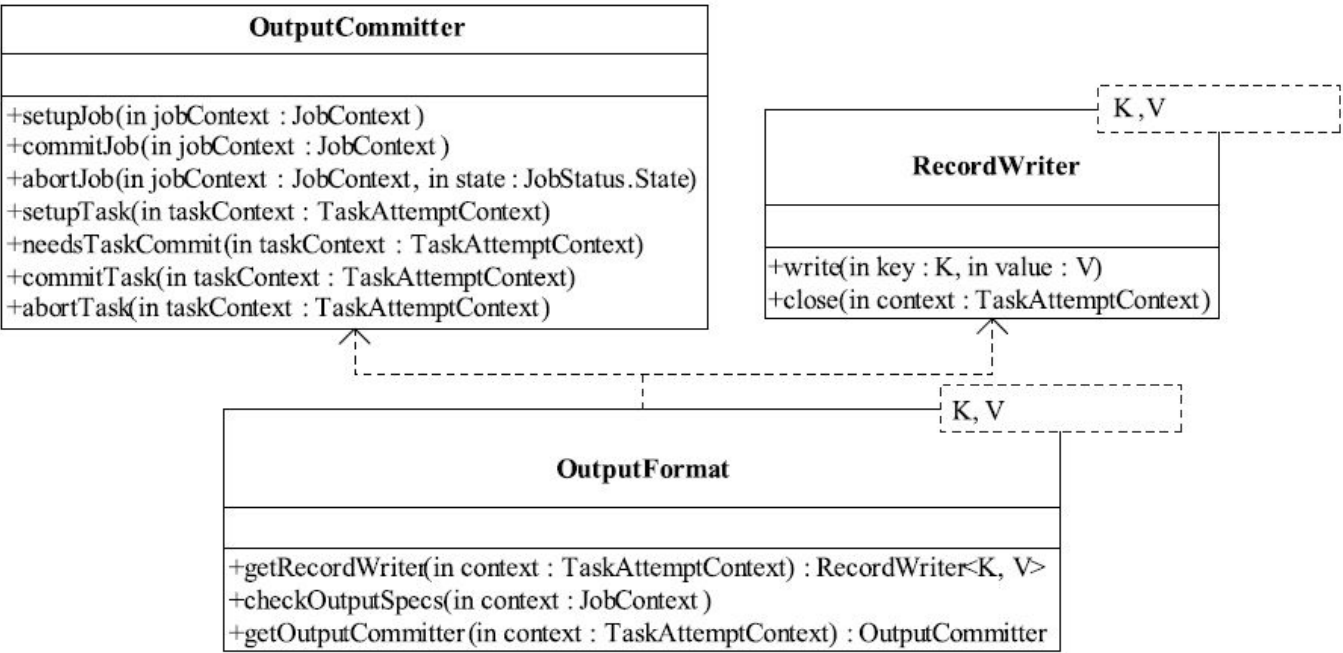


图 3-14 新版API的OutputFormat类图

[1] 具体可到<http://incubator.apache.org/oozie/>下查看相关文档。

3.3.4 Mapper与Reducer解析

1.旧版API的Mapper/Reducer解析

Mapper/Reducer中封装了应用程序的数据处理逻辑。为了简化接口，MapReduce要求所有存储在底层分布式文件系统上的数据均要解释成key/value的形式，并交给Mapper/Reducer中的map/reduce函数处理，产生另外一些key/value。

Mapper与Reducer的类体系非常类似，我们以Mapper为例进行讲解。Mapper的类图如图3-15所示，包括初始化、Map操作和清理三部分。

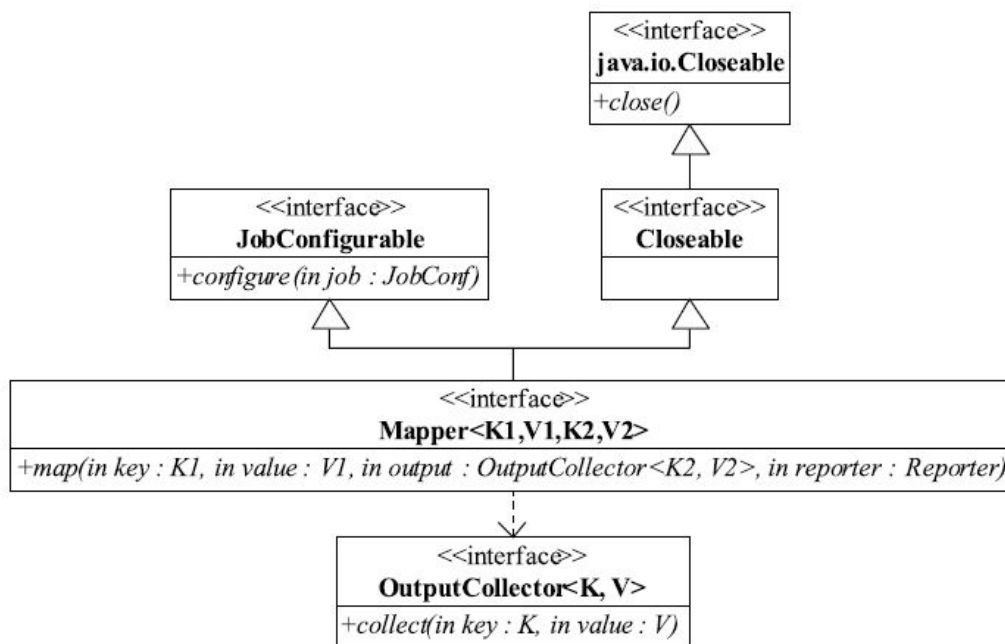


图 3-15 旧版API的Mapper类图

(1) 初始化

Mapper继承了JobConfigurable接口。该接口中的configure方法允许通过JobConf参数对Mapper进行初始化。

(2) Map操作

MapReduce框架会通过InputFormat中RecordReader从InputSplit获取一个个key/value对，并交给下面的map()函数处理：

```
void map (K1 key, V1 value, OutputCollector<K2, V2>output, Reporter reporter) throws IOException;
```

该函数的参数除了key和value之外，还包括OutputCollector和Reporter两个类型的参数，分别用于输出结果和修改Counter值。

(3) 清理

Mapper通过继承Closeable接口（它又继承了Java IO中的Closeable接口）获得close方法，用户可通过实现该方法对Mapper进行清理。

MapReduce提供了很多Mapper/Reducer实现，但大部分功能比较简单，具体如图3-16所示。它们对应的功能分别是：

- ChainMapper/ChainReducer: 用于支持链式作业，具体见3.5.2节。
- IdentityMapper/IdentityReducer: 对于输入key/value不进行任何处理，直接输出。
- InvertMapper: 交换key/value位置。
- RegexMapper: 正则表达式字符串匹配。

- ❑TokenMapper: 将字符串分割成若干个token（单词），可用作WordCount的Mapper。
- ❑LongSumReducer: 以key为组，对long类型的value求累加和。

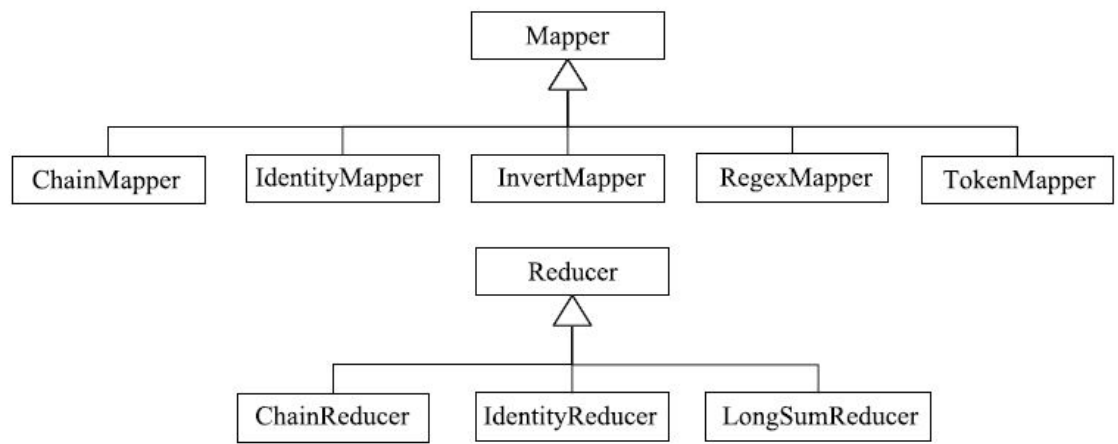


图 3-16 Hadoop MapReduce自带Mapper/Reducer实现的类层次图

对于一个MapReduce应用程序，不一定非要存在Mapper。MapReduce框架提供了比Mapper更通用的接口：MapRunnable，如图3-17所示。用户可以实现该接口以定制Mapper的调用方式或者自己实现key/value的处理逻辑，比如，Hadoop Pipes自行实现了MapRunnable，直接将数据通过Socket发送给其他进程处理。提供该接口的另外一个好处是允许用户实现多线程Mapper。

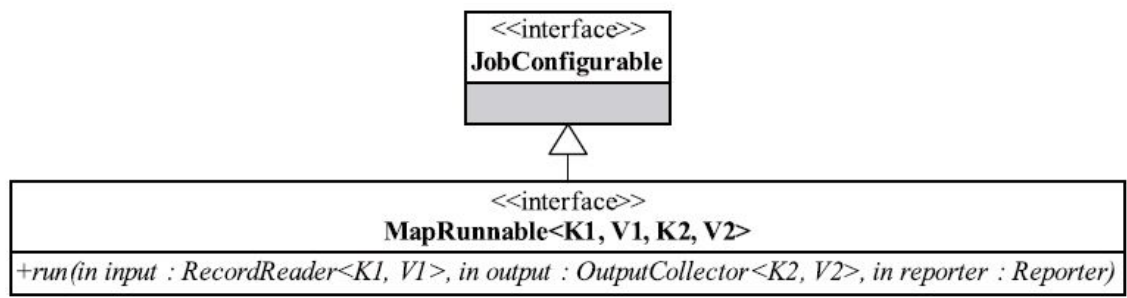


图 3-17 MapRunnable类图

如图3-18所示，MapReduce提供了两个MapRunnable实现，分别是MapRunner和MultithreadedMapRunner，其中MapRunner为默认实现。MultithreadedMapRunner实现了一种多线程的MapRunnable。默认情况下，每个Mapper启动10个线程，通常用于非CPU类型的作业以提供吞吐率。

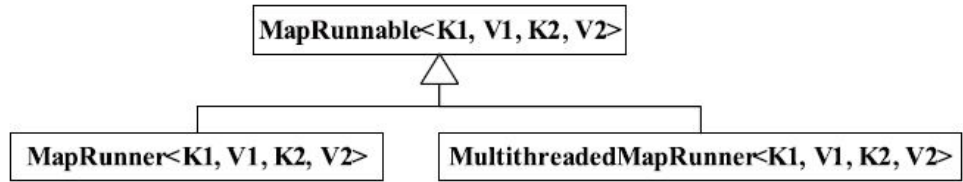


图 3-18 Hadoop MapReduce自带MapRunnable实现的类层次图

2.新版API的Mapper/Reducer解析

从图3-19可知，新API在旧API基础上发生了以下几个变化：

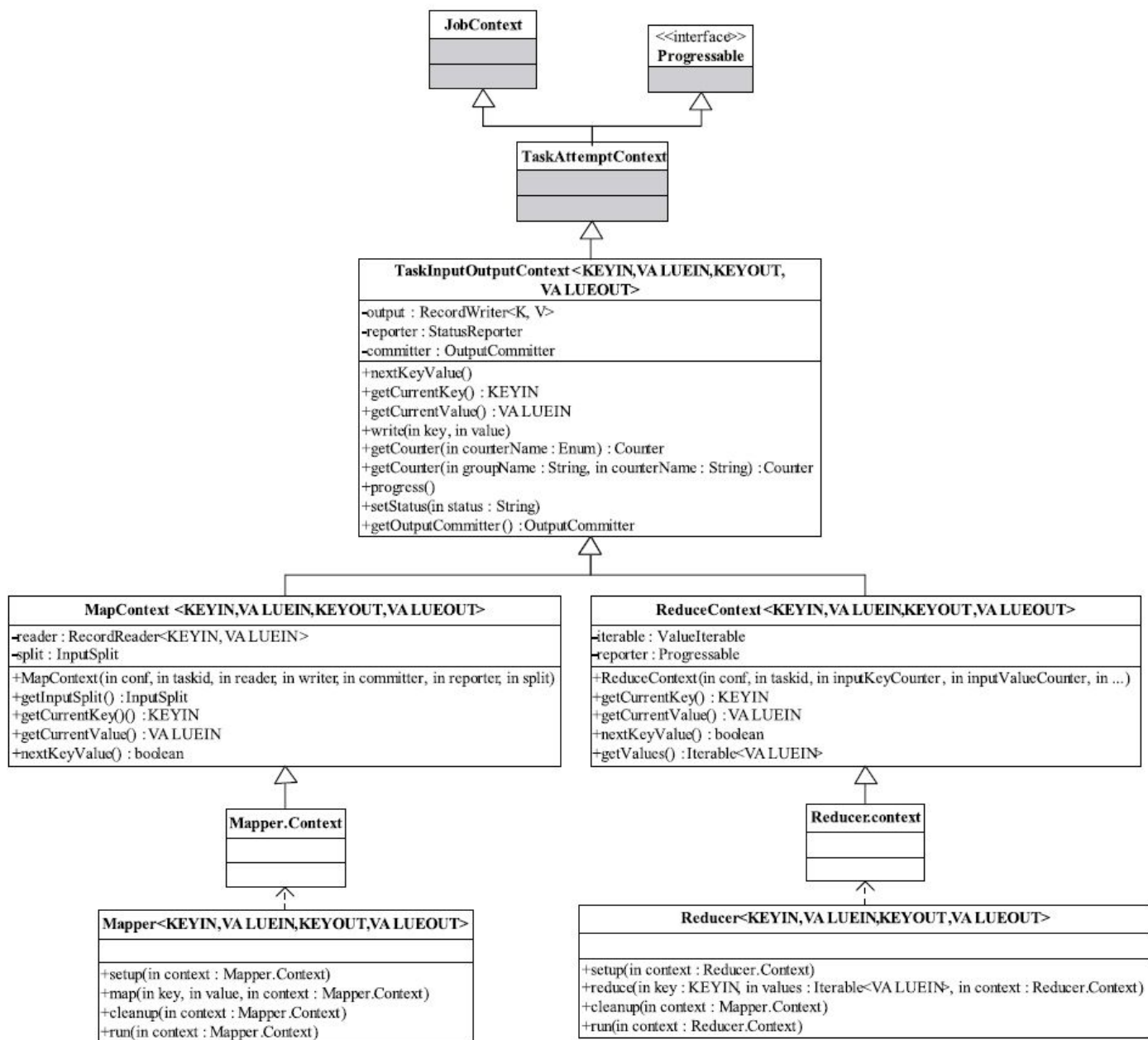


图 3-19 新版API的Mapper/Reducer类图

□ Mapper由接口变为抽象类，且不再继承JobConfigurable和Closeable两个接口，而是直接在类中添加了setup和cleanup两个方法进行初始化和清理工作。

□ 将参数封装到Context对象中，这使得接口具有良好的扩展性。

□ 去掉MapRunnable接口，在Mapper中添加run方法，以方便用户定制map()函数的调用方法，run默认实现与旧版本中MapRunner的run实现一样。

□ 新API中Reducer遍历value的迭代器类型变为java.lang.Iterable，使得用户可以采用“foreach”形式遍历所有value，如下所示：

```

void reduce (KEYIN key, Iterable<VALUEIN>values, Context context
) throws IOException, InterruptedException{
    for (VALUEIN value: values) { //注意遍历方式
        context.write ( (KEYOUT) key, (VALUEOUT) value);
    }
}
  
```

3.3.5 Partitioner接口的设计与实现

Partitioner的作用是对Mapper产生的中间结果进行分片，以便将同一分组的数据交给同一个Reducer处理，它直接影响Reduce阶段的负载均衡。旧版API中Partitioner的类图如图3-20所示。它继承了JobConfigurable，可通过configure方法初始化。它本身只包含一个待实现的方法getPartition。该方法包含三个参数，均由框架自动传入，前面两个参数是key/value，第三个参数numPartitions表示每个Mapper的分片数，也就是Reducer的个数。

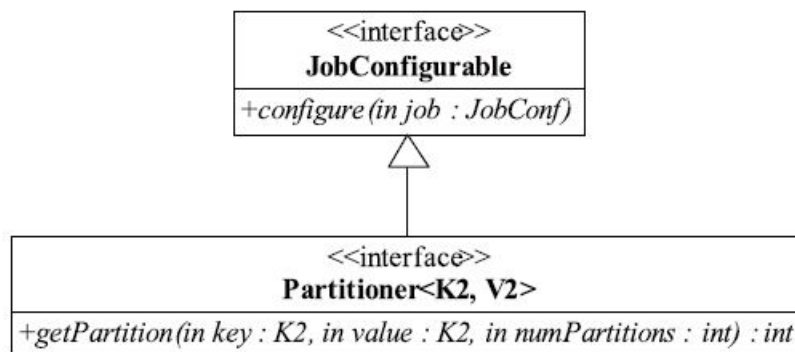


图 3-20 旧版API的Partitioner类图

MapReduce提供了两个Partitioner实现：HashPartitioner和TotalOrderPartitioner。其中HashPartitioner是默认实现，它实现了一种基于哈希值的分片方法，代码如下：

```
public int getPartition(K2 key, V2 value,
int numReduceTasks) {
return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
}
```

TotalOrderPartitioner提供了一种基于区间的分片方法，通常用在数据全排序中。在MapReduce环境中，容易想到的全排序方案是归并排序，即在Map阶段，每个Map Task进行局部排序；在Reduce阶段，启动一个Reduce Task进行全局排序。由于作业只能有一个Reduce Task，因而Reduce阶段会成为作业的瓶颈。为了提高全局排序的性能和扩展性，MapReduce提供了TotalOrderPartitioner。它能够按照大小将数据分成若干个区间（分片），并保证后一个区间的所有数据均大于前一个区间数据，这使得全排序的步骤如下：

步骤1 数据采样。在Client端通过采样获取分片的分割点。Hadoop自带了几个采样算法，如IntercalSampler、RandomSampler、SplitSampler等（具体见org.apache.hadoop.mapred.lib包中的InputSampler类）。下面举例说明。

采样数据为：b, abc, abd, bcd, abcd, efg, hii, afd, rrr, mnk

经排序后得到：abc, abcd, abd, afd, b, bcd, efg, hii, mnk, rrr

如果Reduce Task个数为4，则采样数据的四等分点为abd、bcd、mnk，将这3个字符串作为分割点。

步骤2 Map阶段。本阶段涉及两个组件，分别是Mapper和Partitioner。其中，Mapper可采用IdentityMapper，直接将输入数据输出，但Partitioner必须选用TotalOrderPartitioner，它将步骤1中获取的分割点保存到trie树中以便快速定位任意一个记录所在的区间，这样，每个Map Task产生R（Reduce Task个数）个区间，且区间之间有序。

TotalOrderPartitioner通过trie树查找每条记录所对应的Reduce Task编号。如图3-21所示，我们将分割点保存在深度为2的trie树中，假设输入数据中有两个字符串“abg”和“mnz”，则字符串“abg”对应partition1，即第2个Reduce Task，字符串“mnz”对应partition3，即第4个Reduce Task。

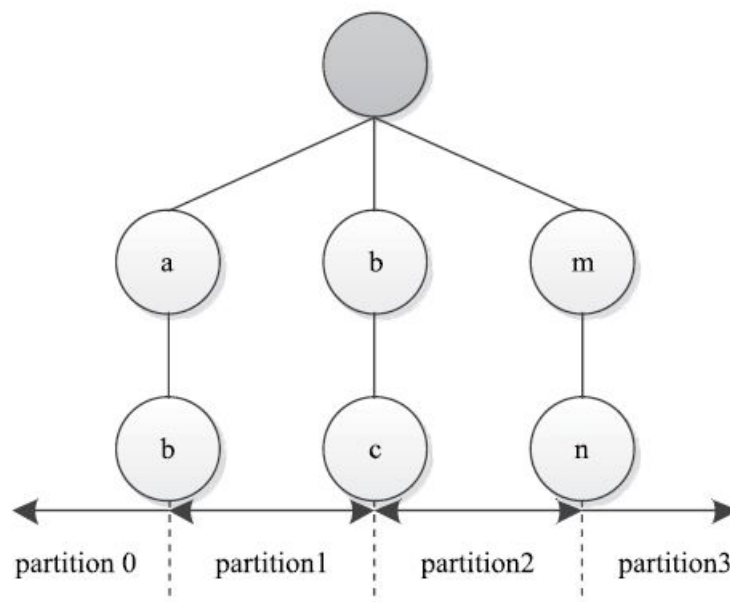


图 3-21 利用trie树对数据进行分片

步骤3 Reduce阶段。每个Reducer对分配到的区间数据进行局部排序，最终得到全排序数据。

从以上步骤可以看出，基于TotalOrderPartitioner全排序的效率跟key分布规律和采样算法有直接关系；key值分布越均匀且采样越具有代表性，则Reduce Task负载越均衡，全排序效率越高。

TotalOrderPartitioner有两个典型的应用实例：TeraSort和HBase批量数据导入。其中，TeraSort是Hadoop自带的一个应用程序实例。它曾在TB级数据排序基准评估中赢得第一名^[1]，而TotalOrderPartitioner正是从该实例中提炼出来的。HBase^[2]是一个构建在Hadoop之上的NoSQL数据仓库。它以Region为单位划分数据，Region内部数据有序（按key排序），Region之间也有序。很明显，一个MapReduce全排序作业的R个输出文件正好可对应HBase的R个Region。

新版API中的Partitioner类图如图3-22所示。它不再实现JobConfigurable接口。当用户需要让Partitioner通过某个JobConf对象初始化时，可自行实现Configurable接口，如：

```
public class TotalOrderPartitioner<K, V>
    extends Partitioner<K, V> implements Configurable
```

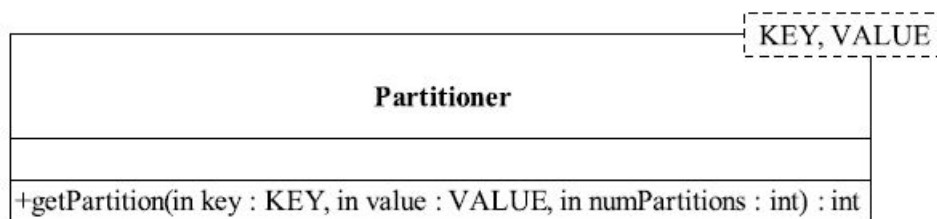


图 3-22 新版API中的Partitioner类图

[1] <http://sortbenchmark.org/>

[2] <http://hbase.apache.org/>

3.4 非Java API解析

3.4.1 Hadoop Streaming的实现原理

Hadoop Streaming是Hadoop为方便非Java用户编写MapReduce程序而设计的工具包。它允许用户将任何可执行文件或者脚本作为Mapper/Reducer，这大大提高了程序员的开发效率。

Hadoop Streaming要求用户编写的Mapper/Reducer从标准输入中读取数据，并将结果写到标准数据中，这类似于Linux中的管道机制。

1.Hadoop Streaming编程实例

以WordCount为例，可用C++分别实现Mapper和Reducer，具体方法如下（这里仅是最简单的实现，并未全面考虑各种异常情况）。

Mapper实现的具体代码如下：

```
int main() { //Mapper将会被封装成一个独立进程，因而需要有main() 函数
string key;
while (cin >> key) { //从标准输入流中读取数据
//输出中间结果，默认情况下TAB为key/value分隔符
cout << key << "\t" << "1" << endl;
}
return 0;
}
```

Reducer实现的具体代码如下：

```
int main() { //Reducer将会被封装成一个独立进程，因而需要有main() 函数
string cur_key, last_key, value;
cin >> cur_key >> value;
last_key = cur_key;
int n = 1;
while (cin >> cur_key) { //读取Map Task输出结果
cin >> value;
if (last_key != cur_key) { //识别下一个key
cout << last_key << "\t" << n << endl;
last_key = cur_key;
n = 1;
} else { //获取key相同的所有value数目
n++; //key值相同的，累计value值
}
}
return 0;
}
```

分别编译这两个程序，生成的可执行文件分别是wc_mapper和wc_reducer，并将它们和contrib/streaming/hadoop-streaming-1.0.0.jar一起复制到Hadoop安装目录下，使用以下命令提交作业：

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming-1.0.0.jar \
-files wc_mapper, wc_reducer \
-input/test/input \
-output/test/output \
-mapper wc_mapper \
-reducer wc_reducer
```

由于Hadoop Streaming类似于Linux管道，这使得测试变得非常容易。用户可直接在本地使用下面命令测试结果是否正确：

```
cat test.txt | ./wc_mapper | sort | ./wc_reducer
```

2.Hadoop Streaming实现原理分析

Hadoop Streaming工具包实际上是一个使用Java编写的MapReduce作业。当用户使用可执行文件或者脚本文件充当Mapper或者

Reducer时，Java端的Mapper或者Reducer充当了wrapper角色，它们将输入文件中的key和value直接传递给可执行文件或者脚本文件进行处理，并将处理结果写入HDFS。

实现Hadoop Streaming的关键技术点是如何使用标准输入输出实现Java与其他可执行文件或者脚本文件之间的通信。为此，Hadoop Streaming使用了JDK中的java.lang.ProcessBuilder类。该类提供了一整套管理操作系统进程的方法，包括创建、启动和停止进程（也就是应用程序）等。相比于JDK中的Process类，ProcessBuilder允许用户对进程进行更多控制，包括设置当前工作目录、改变环境参数等。

下面分析Mapper的执行过程（Reducer的类似）。整个过程如图3-23所示，Hadoop Streaming使用ProcessBuilder以独立进程方式启动可执行文件wc_mapper，并创建该进程的输入输出流，具体实现代码如下：

```
.....
//将wc_mapper封装成一个进程
ProcessBuilder builder=new ProcessBuilder("wc_mapper");
builder.environment().putAll(childEnv.toMap()); //设置环境变量
sim=builder.start();
//创建标准输出流
clientOut_=new DataOutputStream(new BufferedOutputStream(
sim.getOutputStream(),
BUFFER_SIZE));
//创建标准输入流
clientIn_=new DataInputStream(new BufferedInputStream(
sim.getInputStream(),
BUFFER_SIZE));
//创建标准错误流
clientErr_=new DataInputStream(new
BufferedInputStream(sim.getErrorStream()));
```

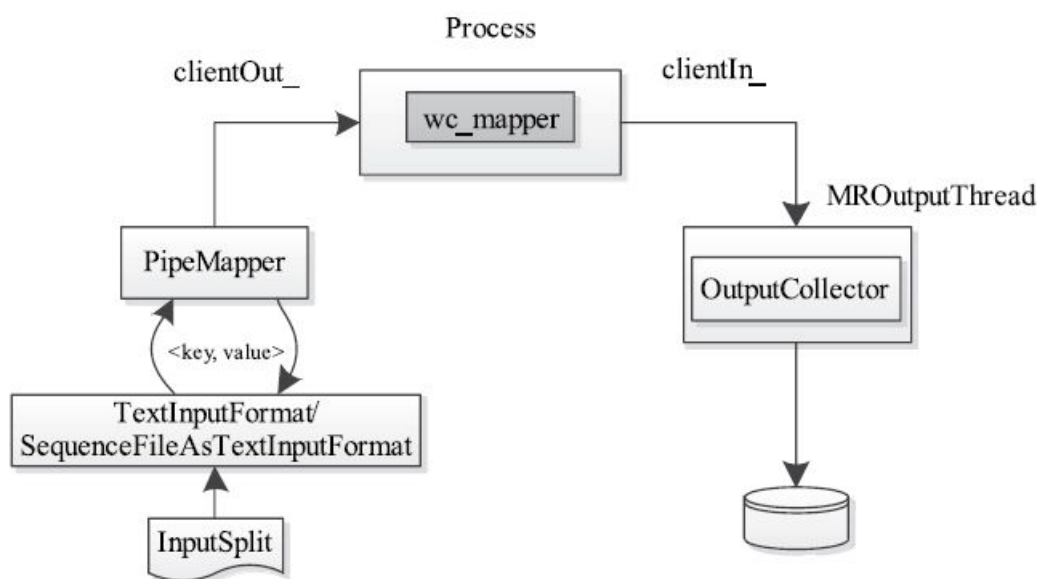


图 3-23 Hadoop Streaming工作原理图

Hadoop Streaming提供了一个默认的PipeMapper。它实际上是C++端Mapper的wrapper，主要作用是向已经创建好的输出流clientOut_中写入数据，具体实现代码如下：

```
public void map(Object key, Object value, OutputCollector output, Reporter
reporter) throws IOException{
.....
clientOut_.write(key, 0, keySize);
clientOut_.write(mapInputFieldSeparator);
clientOut_.write(value, 0, valueSize);
clientOut_.write('\n');
}
```

写入clientOut_的数据直接成为wc_mapper的输入，待数据被处理完后，可直接从标准输入流clientIn_中获取结果：

```
//MROutputThread
```

```
public void run(){
    lineReader=new LineReader ((InputStream) clientIn_, job_);
    while (lineReader.readLine (line) >0) {
        splitKeyVal (line, line.getLength(), key, val);
        output.collect (key, val);
    }
}
```

通过分析以上代码可知，由于Hadoop Streaming使用分隔符定位一个完整的key或value，因而只能支持文本格式数据，不支持二进制格式。在0.21.0/0.22.X系列版本中，Hadoop Streaming增加了对二进制文件的支持^[1]，并添加了两种新的二进制文件格式：RawBytes和TypedBytes。顾名思义，RawBytes指key和value是原始字节序列，而TypedBytes指key和value可以拥有的数据类型，比如boolean、list、map等。由于它们采用的是长度而不是某一种分隔符定位key和value，因而支持二进制文件格式。

RawBytes传递给可执行文件或者脚本文件的内容编码格式为：

```
<4 byte length><key raw bytes><4 byte length><value raw bytes>
```

TypedBytes允许用户为key和value指定数据类型。对于长度固定的基本类型，如byte、bool、int、long等，其编码格式为：

```
<1 byte type code><key bytes><1 byte type code><value bytes>
```

对于长度不固定的类型，如byte array、string等，其编码格式为：

```
<1 byte type code><4 byte length><key raw bytes><1 byte type code><4 byte length><value raw bytes>
```

当key和value大部分情况下为固定长度的基本类型时，TypedBytes比RawBytes格式更节省空间。

^[1] <https://issues.apache.org/jira/browse/HADOOP-1722>

3.4.2 Hadoop Pipes的实现原理

Hadoop Pipes是Hadoop为方便C/C++用户编写MapReduce程序而设计的工具。其设计思想是将应用逻辑相关的C++代码放在单独的进程中，然后通过Socket让Java代码与C++代码通信以完成数据计算。

1.编程实例

同样，以WordCount为例，采用C++分别编写Mapper和Reducer，具体方法如下。

Mapper实现的具体代码如下：

```
class WordCountMapper: public HadoopPipes: Mapper{//注意基类
public:
WordCountMapper (HadoopPipes: TaskContext&context) {
//在此初始化，比如定义计数器等
}
//MapContext封装了Mapper的各种操作
void map (HadoopPipes: MapContext&context) {
std: vector<std: string>words=
HadoopUtils: splitString (context.getInputValue(), "");
for (unsigned int i=0; i<words.size(); ++i) {
context.emit (words[i], "1"); //用emit输出key/value对
}
}
};
```

Reducer实现的具体代码如下：

```
class WordCountReducer: public HadoopPipes: Reducer{
public:
WordCountReducer (HadoopPipes: TaskContext&context) {
}
//ReduceContext封装了Reducer的各种操作
void reduce (HadoopPipes: ReduceContext&context) {
int sum=0;
while (context.nextValue()) { //迭代获取该key对应的value列表
sum+=HadoopUtils: toInt (context.getInputValue());
}
context.emit (context.getInputKey(), HadoopUtils: toString (sum));
}
};
```

main()函数的具体实现代码如下：

```
//每个Hadoop Pipes作业将被单独封装成一个进程，因此需要有main()函数
int main (int argc, char*argv[]) {
return HadoopPipes: runTask (
HadoopPipes: TemplateFactory<WordCountMap, WordCountReduce>());
}
```

编译之后生成可执行文件wordcount，输入以下命令运行作业：

```
bin/hadoop pipes\
-D hadoop.pipes.java.recordreader=true\
-D hadoop.pipes.java.recordwriter=true\
-D mapred.job.name=wordcount\
-input/test/intput\
-output/test/output\
-program wordcount
```

与Hadoop Streaming比较，可以发现，Hadoop Pipes的一个缺点是调试不方便。因为输入的数据是Java端代码通过Socket传到C++应用程序的，所以用户不能单独对C++部分代码进行测试，而需要连同Java端代码一起启动。

2.实现原理分析

Hadoop Pipes的实现原理与Hadoop Streaming非常类似，它也使用Java中的ProcessBuilder以单独进程方式启动可执行文件。不同之处是Java代码与可执行文件（或者脚本）的通信方式：Hadoop Streaming采用标准输入输出，而Hadoop Pipes采用Socket。

Hadoop Pipes由两部分组成：Java端代码和C++端代码。与Hadoop Streaming一样，Java端代码实际上实现了一个MapReduce作业，Java端的Mapper或者Reducer实际上是C++端Mapper或者Reducer的封装器（wrapper），它们通过Socket将输入的key和value直接传递给可执行文件执行。

Hadoop Pipes具体执行流程如图3-24所示。该序列图阐释了执行Mapper时，Java端与C++端通过Socket进行交互的过程，主要有以下几个步骤：

- 步骤1 用户提交Pipes作业后，Java端启动一个Socket server（等待C++端接入），同时以独立进程方式运行C++端代码。
- 步骤2 C++端以Client身份连接Java端的Socket server，连接成功后，Java端依次发送一系列指令通知C++端进行各项准备工作。
- 步骤3 Java端通过mapItem()函数不断向C++端传送key/value对，C++端将计算结果返回给Java端，Java端对结果进行保存。
- 步骤4 所有数据处理完毕后，Java端通知C++端终止计算，并关闭C++端进程。

上面分析了Java端与C++端的交互过程，接下来深入分析Hadoop Pipes内部实现原理。如图3-25所示，Java端用PipesMapRunner实现了MapRunner，在MapRunner内部，借助两个协议类DownwardProtocol和UpwardProtocol向C++端发送数据和从C++端接收数据，而C++端也有两个类与之对应，分别是Protocol和UpwardProtocol。Protocol将收到的数据传给用户编写的Mapper，经Mapper、Combiner和Partitioner处理后，由UpwardProtocol返回给Java端的UpwardProtocol，由它写到本地磁盘上。

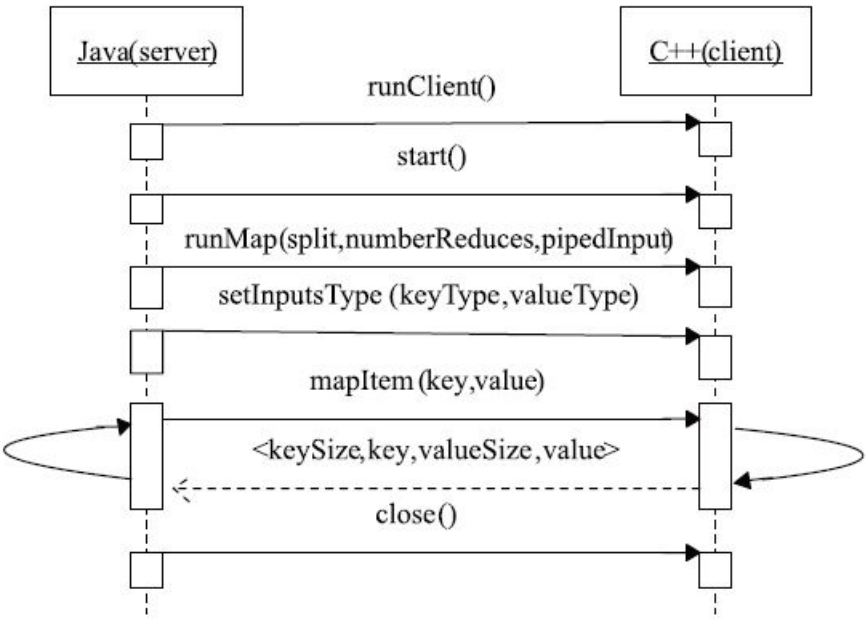


图 3-24 Hadoop Pipes中Java端与C++端交互序列图

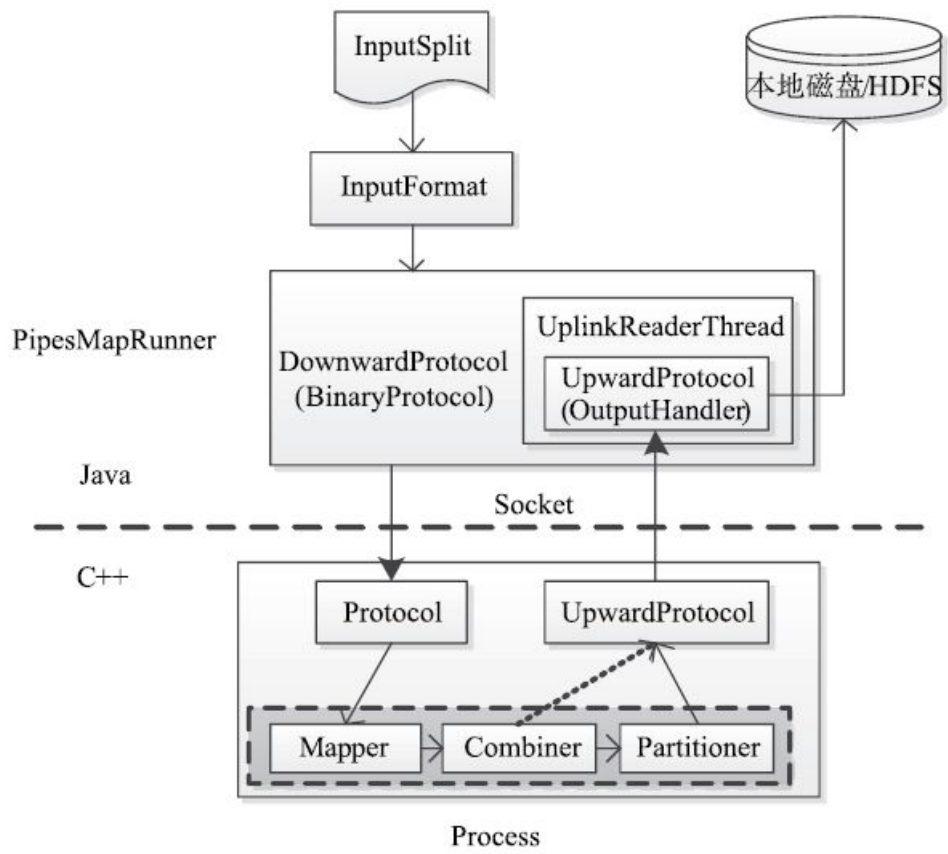


图 3-25 Hadoop Pipes内部实现原理图

3.5 Hadoop工作流

前面仅是介绍了单一作业的编写方法，很多情况下，用户编写的作业比较复杂，相互之间存在依赖关系，这种依赖关系可以用有向图表示，我们称之为“工作流”。本节将介绍Hadoop工作流的编写方法、设计原理以及实现。

3.5.1 JobControl的实现原理

1.JobControl编程实例

我们以第2章中的贝叶斯分类为例介绍。一个完整的贝叶斯分类算法可能需要4个有依赖关系的MapReduce作业完成，传统的做法是：为每个作业创建相应的JobConf对象，并按照依赖关系依次（串行）提交各个作业，如下所示：

```
//为4个作业分别创建JobConf对象
JobConf extractJobConf=new JobConf (ExtractJob.class);
JobConf classPriorJobConf=new JobConf (ClassPriorJob.class);
JobConf conditionalProbabilityJobConf=new JobConf (ConditionalProbabilityJob.class);
JobConf predictJobConf=new JobConf (PredictJob.class);
.....//配置各个JobConf
//按照依赖关系依次提交作业
JobClient.runJob (extractJobConf);
JobClient.runJob (classPriorJobConf);
JobClient.runJob (conditionalProbabilityJobConf);
JobClient.runJob (predictJobConf);
```

如果使用JobControl，则用户只需使用addDepending()函数添加作业依赖关系接口，JobControl会按照依赖关系调度各个作业，具体代码如下：

```
Configuration extractJobConf=new Configuration();
Configuration classPriorJobConf=new Configuration();
Configuration conditionalProbabilityJobConf=new Configuration();
Configuration predictJobConf=new Configuration();
.....//设置各个Configuration
//创建Job对象。注意，JobControl要求作业必须封装成Job对象
Job extractJob=new Job (extractJobConf);
Job classPriorJob=new Job (classPriorJobConf);
Job conditionalProbabilityJob=new Job (conditionalProbabilityJobConf);
Job predictJob=new Job (predictJobConf);
//设置依赖关系，构造一个DAG作业
classPriorJob.addDepending (extractJob);
conditionalProbabilityJob.addDepending (extractJob);
predictJob.addDepending (classPriorJob);
predictJob.addDepending (conditionalProbabilityJob);
//创建JobControl对象，由它对作业进行监控和调度
JobControl JC=new JobControl ("Native Bayes");
JC.addJob (extractJob); //把4个作业加入JobControl中
JC.addJob (classPriorJob);
JC.addJob (conditionalProbabilityJob);
JC.addJob (predictJob);
JC.run (); //提交DAG作业
```

在实际运行过程中，不依赖于其他任何作业的extractJob会优先得到调度，一旦运行完成，classPriorJob和conditionalProbabilityJob两个作业同时被调度，待它们全部运行完成后，predictJob被调度。

对比以上两种方案，可以得到一个简单的结论：使用JobControl编写DAG作业更加简便，且能使多个无依赖关系的作业并行运行。

2.JobControl设计原理分析

JobControl由两个类组成：Job和JobControl。其中，Job类封装了一个MapReduce作业及其对应的依赖关系，主要负责监控各个依赖作业的运行状态，以此更新自己的状态，其状态转移图如图3-26所示。作业刚开始处于WAITING状态。如果没有依赖作业或者所有依赖作业均已运行完成，则进入READY状态。一旦进入READY状态，则作业可被提交到Hadoop集群上运行，并进入RUNNING状态。在RUNNING状态下，根据作业运行情况，可能进入SUCCESS或者FAILED状态。需要注意的是，如果一个作业

的依赖作业失败，则该作业也会失败，于是形成“多米诺骨牌效应”，后续所有作业均会失败。

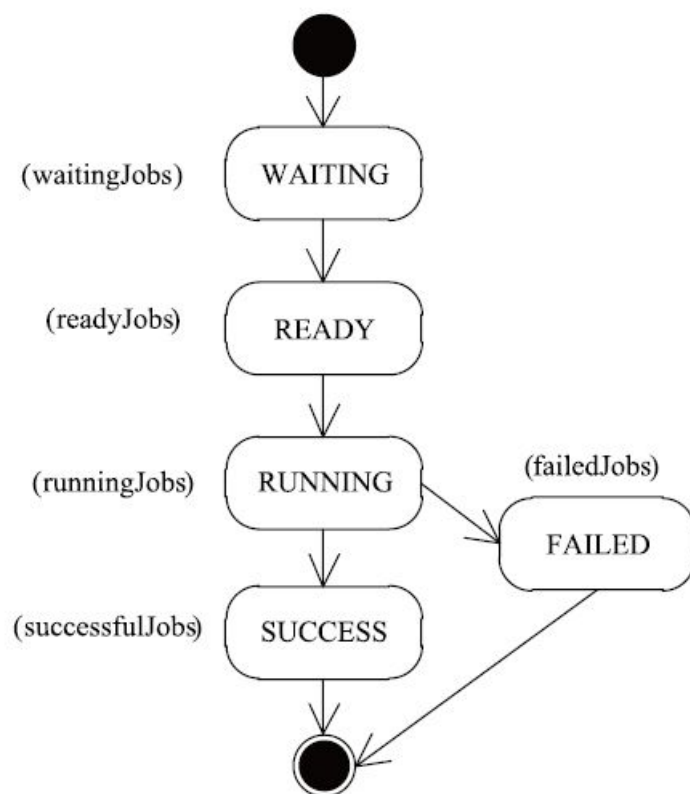


图 3-26 JobControl中Job状态转移图

JobControl封装了一系列MapReduce作业及其对应的依赖关系。它将处于不同状态的作业放入不同的哈希表中，并按照图3-26所示的状态转移作业，直到所有作业运行完成。在实现的时候，JobControl包含一个线程用于周期性地监控和更新各个作业的运行状态，调度依赖作业运行完成的作业，提交处于READY状态的作业等。同时，它还提供了一些API用于挂起、恢复和暂停该线程。

3.5.2 ChainMapper/ChainReducer的实现原理

ChainMapper/ChainReducer主要为了解决线性链式Mapper而提出的。也就是说，在Map或者Reduce阶段存在多个Mapper，这些Mapper像Linux管道一样，前一个Mapper的输出结果直接重定向到下一个Mapper的输入，形成一个流水线，形式类似于[MAP+REDUCE MAP*]。图3-27展示了一个典型的ChainMapper/ChainReducer的应用场景：在Map阶段，数据依次经过Mapper1和Mapper2处理；在Reduce阶段，数据经过shuffle和sort后，交由对应的Reducer处理，但Reducer处理之后并没有直接写到HDFS上，而是交给另外一个Mapper处理，它产生的结果写到最终的HDFS输出目录中。

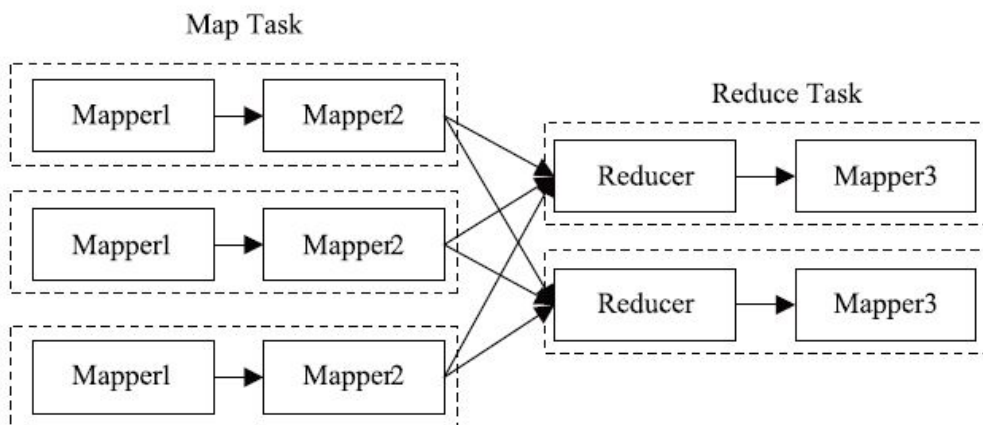


图 3-27 ChainMapper/ChainReducer应用实例

需要注意的是，对于任意一个MapReduce作业，Map和Reduce阶段可以有无限个Mapper，但Reducer只能有一个。也就是说，图3-28所示的计算过程不能使用ChainMapper/ChainReducer完成，而需要分解成两个MapReduce作业。

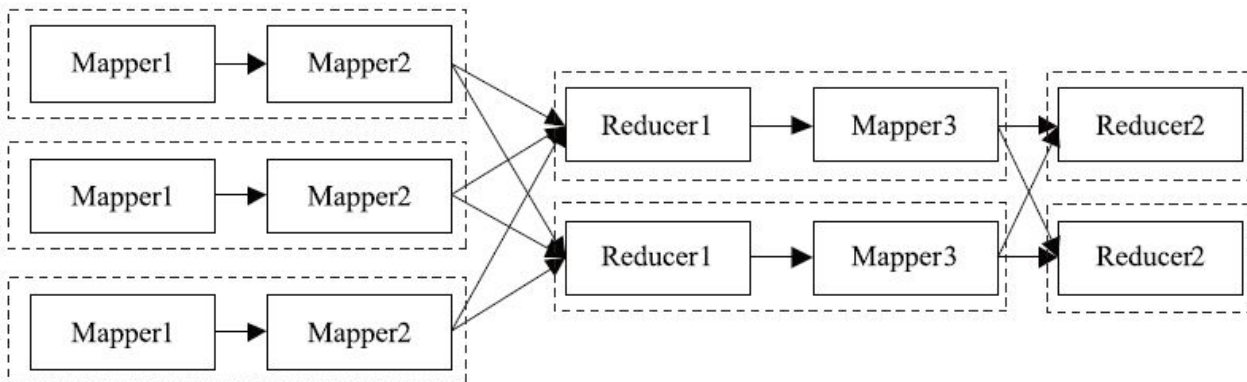


图 3-28 一个ChainMapper/ChainReducer不适用的场景

1.编程实例

这里以图3-27中的作业为例，给出ChainMapper/ChainReducer的基本使用方法，具体代码如下：

```
.....
conf.setJobName("chain");
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);
JobConf mapper1Conf=new JobConf(false);
JobConf mapper2Conf=new JobConf(false);
JobConf reduce1Conf=new JobConf(false);
JobConf mapper3Conf=new JobConf(false);
.....
ChainMapper.addMapper(conf, Mapper1.class, LongWritable.class, Text.class, Text.class, Text.class, true,
mapper1Conf);
ChainMapper.addMapper(conf, Mapper2.class, Text.class, Text.class, LongWritable.class, Text.class, false,
mapper2Conf);
ChainReducer.setReducer(conf, Reducer.class, LongWritable.class, Text.class, Text.class, Text.class, true,
reduce1Conf);
ChainReducer.addMapper(conf, Mapper3.class, Text.class, Text.class, LongWritable.class, Text.class, false,
```

```

null);
    JobClient.runJob (conf);

```

用户通过addMapper在Map/Reduce阶段添加多个Mapper。该函数带有8个输入参数，分别是作业的配置、Mapper类、Mapper的输入key类型、输入value类型、输出key类型、输出value类型、key/value是否按值传递和Mapper的配置。其中，第7个参数需要解释一下：Hadoop MapReduce有一个约定，函数OutputCollector.collect（key, value）执行期间不应改变key和value的值。这主要是因为函数Mapper.map()调用完OutputCollector.collect（key, value）之后，可能会再次使用key和value值，如果被改变，可能会造成潜在的错误。为了防止OutputCollector直接对key/value修改，ChainMapper允许用户指定key/value传递方式。如果用户确定key/value不会被修改，则可选用按引用传递，否则按值传递。需要注意的是，引用传递可避免对象拷贝，提高处理效率，但需要确保key/value不会被修改。

2.实现原理分析

ChainMapper/ChainReducer实现的关键技术点是修改Mapper和Reducer的输出流，将本来要写入文件的输出结果重定向到另外一个Mapper中。在3.3.4节中提到，结果的输出由OutputCollector管理，因而，ChainMapper/ChainReducer需要重新实现一个OutputCollector完成数据重定向功能。

尽管链式作业在Map和Reduce阶段添加了多个Mapper，但仍然只是一个MapReduce作业，因而只能有一个与之对应的JobConf对象。然而，当用户调用addMapper添加Mapper时，可能会为新添加的每个Mapper指定一个特有的JobConf，为此，ChainMapper/ChainReducer将这些JobConf对象序列化后，统一保存到作业的JobConf中。图3-27中的实例可能产生如表3-3所示的几个配置选项。

表 3-3 图 3-27 中实例对应的几个配置选项

配置参数	参数值
chain.mapper.mapper.config.0	Mapper1Conf 序列化后的字符串
chain.mapper.mapper.config.1	Mapper2Conf 序列化后的字符串
chain.reducer.reducer.config.0	ReducerConf 序列化后的字符串
chain.reducer.mapper.config.0	Mapper3Conf 序列化后的字符串

当链式作业开始执行的时候，首先将各个Mapper的JobConf对象反序列化，并构造对应的Mapper和Reducer对象，添加到数据结构mappers（List<Mapper>类型）和reducer（Reducer类型）中。ChainMapper中实现的map()函数如下，它调用了第一个Mapper，是后续Mapper的“导火索”。

```

public void map (Object key, Object value, OutputCollector output,
Reporter reporter) throws IOException{
    Mapper mapper=chain.getFirstMap();
    if (mapper!=null) {
        mapper.map (key, value, chain.getMapperCollector (0, output, reporter), reporter);
    }
}

```

chain.getMapperCollector返回一个OutputCollector实现——ChainOutputCollector，它的collect方法如下：

```

public void collect (K key, V value) throws IOException{
    if (nextMapperIndex<mappers.size()) {
        //调用下一个Mapper
        nextMapper.map (key, value,
            new ChainOutputCollector (nextMapperIndex,
            nextKeySerialization,
            nextValueSerialization, output, reporter),
            reporter);
    }else{
        //如果是最后一个Mapper，则直接调用真正的OutputCollector
        output.collect (key, value);
    }
}

```


3.5.3 Hadoop工作流引擎

前面介绍的JobControl和ChainMapper/ChainReducer仅可看作运行工作流的工具。它们只具备最简单的工作流引擎功能，比如工作流描述、简单的作业调度等。为了增强Hadoop支持工作流的能力，在Hadoop之上出现了很多开源的工作流引擎，主要可概括为两类：隐式工作流引擎和显式工作流引擎。

隐式工作流引擎在MapReduce之上添加了一个语言抽象层，允许用户使用更简单的方式编写应用程序，比如SQL、脚本语言等。这样，用户无须关注MapReduce的任何细节，降低了用户的学习成本，并可大大提高开发效率。典型的代表有Hive^[1]、Pig^[2]和Cascading^[3]。它们的架构如图3-29所示，从上往下分为以下三层。

- ❑ 功能描述层：直接面向用户提供了一种简单的应用程序编写方法，比如，Hive使用SQL, Pig使用Pig Latin脚本语言，Cascading提供了丰富的Java API。
- ❑ 作业生成器：作业生成器主要将上层的应用程序转化成一批MapReduce作业。这一批MapReduce存在相互依赖关系，实际上是一个DAG。
- ❑ 调度引擎：调度引擎直接构建于MapReduce环境之上，将作业生成器生成的DAG按照依赖关系提交到MapReduce上运行。



图 3-29 隐式工作流引擎架构图

显式工作流引擎直接面向MapReduce应用程序开发者，提供了一种作业依赖关系描述方式，并能够按照这种描述方式进行作业调度。典型的代表有Oozie和Azkaban^[4]。它们的架构如图3-30所示，从上往下分为以下两层。

- ❑ 工作流描述语言：工作流描述语言用于描述作业的依赖关系。Oozie采用了XML，而Azkaban采用了key/value格式的文本文件。需要注意的是，这里的作业不仅仅是指MapReduce作业，还包括Shell命令、Pig脚本等。也就是说，一个MapReduce可能依赖一个Pig脚本或者Shell命令。
- ❑ 调度引擎：同隐式工作流引擎的调度引擎功能相同，即根据作业的依赖关系完成作业调度。



图 3-30 显式工作流引擎架构图

表3-4对比了显式工作流引擎与Hadoop自带的JobControl的不同之处。尽管它们均用于解决Hadoop工作流调度问题，但是在设计思路、使用方法、应用场景等方面都存在明显的不同。

表 3-4 显式工作流引擎与 JobControl 对比

特性	Oozie/Azkaban	JobControl
工作流描述方式	有专门的描述语言	Java API
执行模型	server-side	client-side
是否可跟踪运行进度	可以，通过界面	不可以
是否需要安装	是	否
是否有重试功能	有，可设置作业失败重试机制	没有

(续)

特性	Oozie/Azkaban	JobControl
依赖关系中是否可有 Pig 脚本、Shell 命令等	可以	不可以，依赖关系只存在于使用 Java 编写的 MapReduce 作业之间

[1] 可参考<http://hive.apache.org/>相关文档。

[2] 可参考<http://pig.apache.org/>相关文档。

[3] 可参考<http://www.cascading.org/>相关文档。

[4] 可参考<http://sna-projects.com/azkaban/>相关文档。

3.6 小结

MapReduce编程模型直接决定了MapReduce的易用性。本章从简单地使用实例、设计原理以及调用时机等方面介绍了MapReduce编程模型中的各个组件。

从整个体系结构上看，整个编程模型位于应用程序层和MapReduce执行器之间，可以分为两层：第一层是最基本的Java API，第二层构建于Java API之上，添加了几个方便用户编写复杂的MapReduce程序和利用其他语言编写MapReduce程序的工具。

Java API分为新旧两套API。新API在旧API基础上封装而来，在易用性和扩展性方面更好。

为了方便用户采用非Java语言编写MapReduce程序，Hadoop提供了Hadoop Streaming和Hadoop Pipes两个工具。它们本质上都是一个MapReduce作业，区别在于Java语言与非Java语言之间的通信机制。

考虑到实际应用中，用户有时不只是编写单一的MapReduce作业，而是存在复杂依赖关系的DAG作业（工作流），Hadoop MapReduce提供了JobControl和ChainMapper/ChainReducer两个工具。

第三部分 MapReduce核心设计篇

本部分内容

Hadoop RPC框架解析

作业提交与初始化过程分析

JobTracker内部实现剖析

TaskTracker内部实现剖析

Task运行过程分析

第4章 Hadoop RPC框架解析

网络通信模块是分布式系统中最底层的模块。它直接支撑了上层分布式环境下复杂的进程间通信（Inter-Process Communication, IPC）逻辑，是所有分布式系统的基础。远程过程调用（Remote Procedure Call, RPC）是一种常用的分布式网络通信协议。它允许运行于一台计算机的程序调用另一台计算机的子程序，同时将网络的通信细节隐藏起来，使得用户无须额外地为这个交互作用编程。由于RPC大大简化了分布式程序开发，因此备受欢迎。

作为一个分布式系统，Hadoop实现了自己的RPC通信协议，它是上层多个分布式子系统（如MapReduce, HDFS, HBase^[1]等）公用的网络通信模块。本章首先从框架设计及实现等方面介绍了Hadoop RPC，接着介绍了该RPC框架在Hadoop MapReduce中的应用。

4.1 Hadoop RPC框架概述

RPC实际上是分布式计算中客户机/服务器（Client/Server）模型的一个应用实例。对于Hadoop RPC而言，它具有以下几个特点。

□透明性：这是所有RPC框架的最根本特征，即当用户在一台计算机的程序调用另外一台计算机上的子程序时，用户自身不应感觉到其间涉及跨机器间的通信，而是感觉像是在执行一个本地调用。

□高性能：Hadoop各个系统（如HDFS, MapReduce）均采用了Master/Slave结构。其中，Master实际上是一个RPC server，它负责处理集群中所有Slave发送的服务请求。为了保证Master的并发处理能力，RPC server应是一个高性能服务器，能够高效地处理来自多个Client的并发RPC请求。

□可控性：JDK中已经自带了一个RPC框架——RMI（Remote Method Invocation，远程方法调用）。之所以不直接使用该框架，主要是因为考虑到RPC是Hadoop最底层、最核心的模块之一，保证其轻量级、高性能和可控性显得尤为重要，而RMI过于重量级且用户可控之处太少（如网络连接、超时和缓冲等均难以定制或者修改）^[2]。

与其他RPC框架一样，Hadoop RPC主要分为四个部分，分别是序列化层、函数调用层、网络传输层和服务器端处理框架，具体实现机制如下：

□序列化层：序列化层的主要作用是将结构化对象转为字节流以便于通过网络进行传输或写入持久存储。在RPC框架中，它主要用于将用户请求中的参数或者应答转化成字节流以便跨机器传输。第3章中已经提到，Hadoop自己实现了序列化框架，一个类只要实现Writable接口，即可支持对象序列化与反序列化。

❑函数调用层：函数调用层的主要功能是定位要调用的函数并执行该函数。HadoopRPC采用Java反射机制与动态代理实现了函数调用，具体参考4.2.1节。

❑网络传输层：网络传输层描述了Client与Server之间消息传输的方式。Hadoop RPC采用了基于TCP/IP的Socket机制，具体参考4.2.2节。

❑服务器端处理框架：服务器端处理框架可被抽象为网络I/O模型。它描述了客户端与服务器端间信息交互的方式。它的设计直接决定着服务器端的并发处理能力。常见的网络I/O模型有阻塞式I/O、非阻塞式I/O、事件驱动I/O等，而Hadoop RPC采用了基于Reactor设计模式的事件驱动I/O模型，具体参考4.2.3节。

Hadoop RPC总体架构如图4-1所示，自下而上可分为两层。第一层是一个基于Java NIO（New IO）实现的客户机/服务器（Client/Server）通信模型。其中，客户端将用户的调用方法及其参数封装成请求包后发送到服务器端。服务器端收到请求包后，经解包、调用函数、打包结果等一系列操作后，将结果返回给服务器端。为了增强Server端的扩展性和并发处理能力，Hadoop RPC采用了基于事件驱动的Reactor设计模式，在具体实现时，用到了JDK提供的各种功能包，主要包括java.nio（NIO）、java.lang.reflect（反射机制和动态代理）、java.net（网络编程库）等。第二层是供更上层程序直接调用的RPC接口，这些接口底层即为客户机/服务器通信模型。

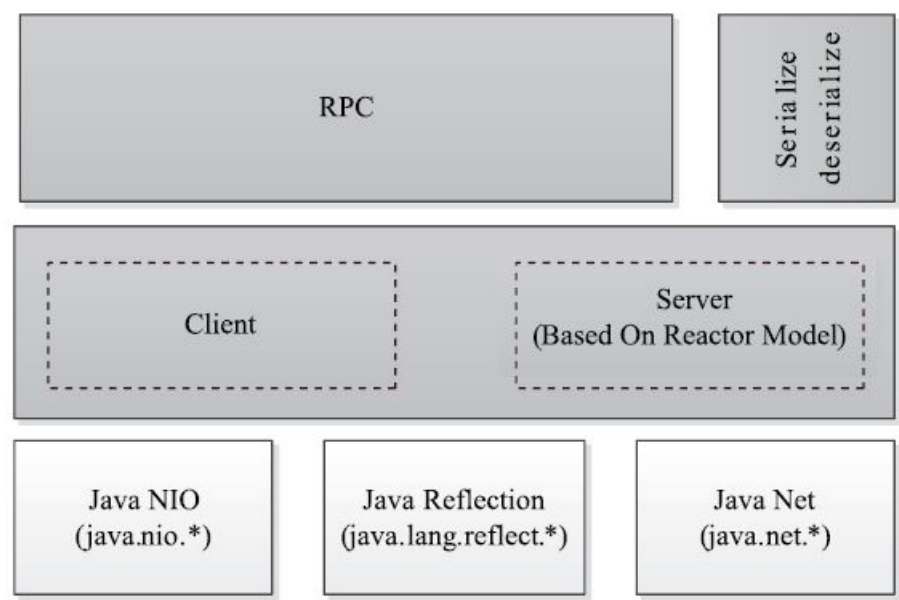


图 4-1 Hadoop RPC总体架构图

[1] HBase RPC在Hadoop RPC基础上做了部分改进。
[2] Doug Cutting在最初设计Hadoop时是这样描述Hadoop RPC设计动机的。

4.2 Java基础知识

在本节中，我们将简要介绍Hadoop RPC中用到的JDK开发工具包中的一些类。了解和掌握这些类的功能和使用方法是深入学习Hadoop RPC的基础。这些类主要来自以下三个Java包：`java.lang.reflect`（反射机制和动态代理相关类）、`java.net`（网络编程库）和`java.nio`（NIO）。

4.2.1 Java反射机制与动态代理

反射机制是Java语言的一个重要特性，它允许用户动态获取类的信息和动态调用对象的方法。它提供的类及类对应的功能如下。

□ `Class`类：代表一个Java类。

□ `Field`类：代表Java类的属性。

□ `Method`类：代表Java类的方法。

□ `Constructor`类：代表Java类的构造函数。

□ `Array`类：提供了动态创建数组，以及访问数组元素的静态方法。

□ `Proxy`类以及`InvocationHandler`接口：提供了动态生成代理类以及实例的方法。

接下来重点介绍Java动态代理。介绍Java动态代理之前，先介绍“代理”这一概念。代理是一种常用的设计模式，其目的是为其他对象提供一种代理以控制对这个对象的访问。代理类负责为委托类进行预处理（如安全检查，权限检查等）或者执行完后的后续处理（如转发给其他代理等）。

Java动态代理机制的实现，使得开发人员通过简单地指定一组接口及委托类对象，便能动态地获得代理类，这大大简化了编写代理类的步骤。

要学习Java动态代理机制，需要首先了解以下几个类或接口。

（1）`java.lang.reflect.Proxy`

这是Java动态代理机制的主类，它提供了一组静态方法，用于为一组接口动态地生成代理类及其对象。`java.lang.reflect.Proxy`的具体实现如下：

```
//方法1：获取指定代理对象所关联的调用处理器
static InvocationHandler getInvocationHandler (Object proxy)
//方法2：获取关联于指定类装载器和一组接口的动态代理类的对象
static Class getProxyClass (ClassLoader loader, Class[] interfaces)
//方法3：判断指定的类对象是否是一个动态代理类
static boolean isProxyClass (Class cl)
//方法4：为指定类装载器、一组接口及调用处理器生成动态代理类实例
static Object newProxyInstance (ClassLoader loader, Class[] interfaces, InvocationHandler h)
```

（2）`java.lang.reflect.InvocationHandler`

这是调用处理器接口。它定义了一个`invoke`方法，用于处理在动态代理类对象上的方法调用。通常开发人员需实现该接口，并在`invoke`方法中实现对委托类的代理访问。`invoke`方法声明如下：

```
//该方法负责处理动态代理类上的所有方法调用。包含三个参数，分别表示代理类实例
//被调用的方法对象、调用参数。调用处理器根据这三个参数进行预处理或分派到委托类实例上执行
```

```
Object invoke (Object proxy, Method method, Object[]args)
```

一个典型的动态代理创建对象的过程可分为以下4个步骤:

步骤1 通过实现InvocationHandler接口创建自己的调用处理器:

```
InvocationHandler handler=new InvocationHandlerImpl (.....);
```

步骤2 通过为Proxy类指定ClassLoader对象和一组interface创建动态代理类:

```
Class clazz=Proxy.getProxyClass (ClassLoader, new Class[]{.....});
```

步骤3 通过反射机制获取动态代理类的构造函数, 其参数类型是调用处理器接口类型:

```
Constructor constructor=clazz.getConstructor (new Class[]{InvocationHandler.class});
```

步骤4 通过构造函数创建动态代理类实例, 此时需将调用处理器对象作为参数被传入:

```
Interface Proxy= (Interface) constructor.newInstance (new Object[]{handler});
```

为了简化对象创建过程, Proxy类中的newInstance方法封装了步骤2~步骤4, 只需两步即可完成代理对象的创建。代码清单4-1给出一个详细的动态代理创建对象的实例: Server类提供了计算整型数相加和相减的两个方法, 用户可直接通过创建代理对象访问这两个方法(类似于RPC Client)。

代码清单4-1 Java动态代理示例程序

```
interface CalculatorProtocol{//定义一个接口协议
public int add(int a, int b); //两个数相加
public int subtract (int a, int b); //两个数相减
}
class Server implements CalculatorProtocol{//实现接口协议
public int add (int a, int b) {
return a+b;
}
public int subtract (int a, int b) {
return a-b;
}
}
//实现调用处理器接口
class CalculatorHandler implements InvocationHandler{
private Object objOriginal;
public CalculatorHandler (Object obj) {
this.objOriginal=obj;
}
public Object invoke (Object proxy, Method method, Object[]args)
throws Throwable{
//可添加一些预处理
Object result=method.invoke (this.objOriginal, args);
//可添加一些后续处理
return result;
}
}
//测试用例
public class DynamicProxyExample{
public static void main (String[]args) {
CalculatorProtocol server=new Server(); //创建Server
InvocationHandler handler=new CalculatorHandler (server);
CalculatorProtocol client= (CalculatorProtocol) Proxy.newProxyInstance (server.
getClass().getClassLoader(), server.getClass().getInterfaces(), handler);
//创建一个Client
int r=client.add (5, 3);
System.out.println ("5+3="+r);
r=client.subtract (10, 2);
System.out.println ("10-2="+r);
}
}
```

4.2.2 Java网络编程

通常，Java网络程序建立在TCP/IP协议基础上，致力于实现应用层。传输层向应用层提供了套接字Socket接口，它封装了底层的数据传输细节；应用层的程序可通过Socket与远程主机建立连接和进行数据传输。

JDK提供了3种套接字类：java.net.Socket、java.net.ServerSocket和java.net.DatagramSocket。其中，java.net.Socket和java.net.ServerSocket类建立在TCP协议基础上，而java.net.DatagramSocket类则建立在UDP协议基础上。Java网络程序均采用客户机/服务器通信模式。下面介绍如何使用java.net.Socket和java.net.ServerSocket编写客户端和服务端程序。

如图4-2所示，编写一个客户端程序需要以下3个步骤。

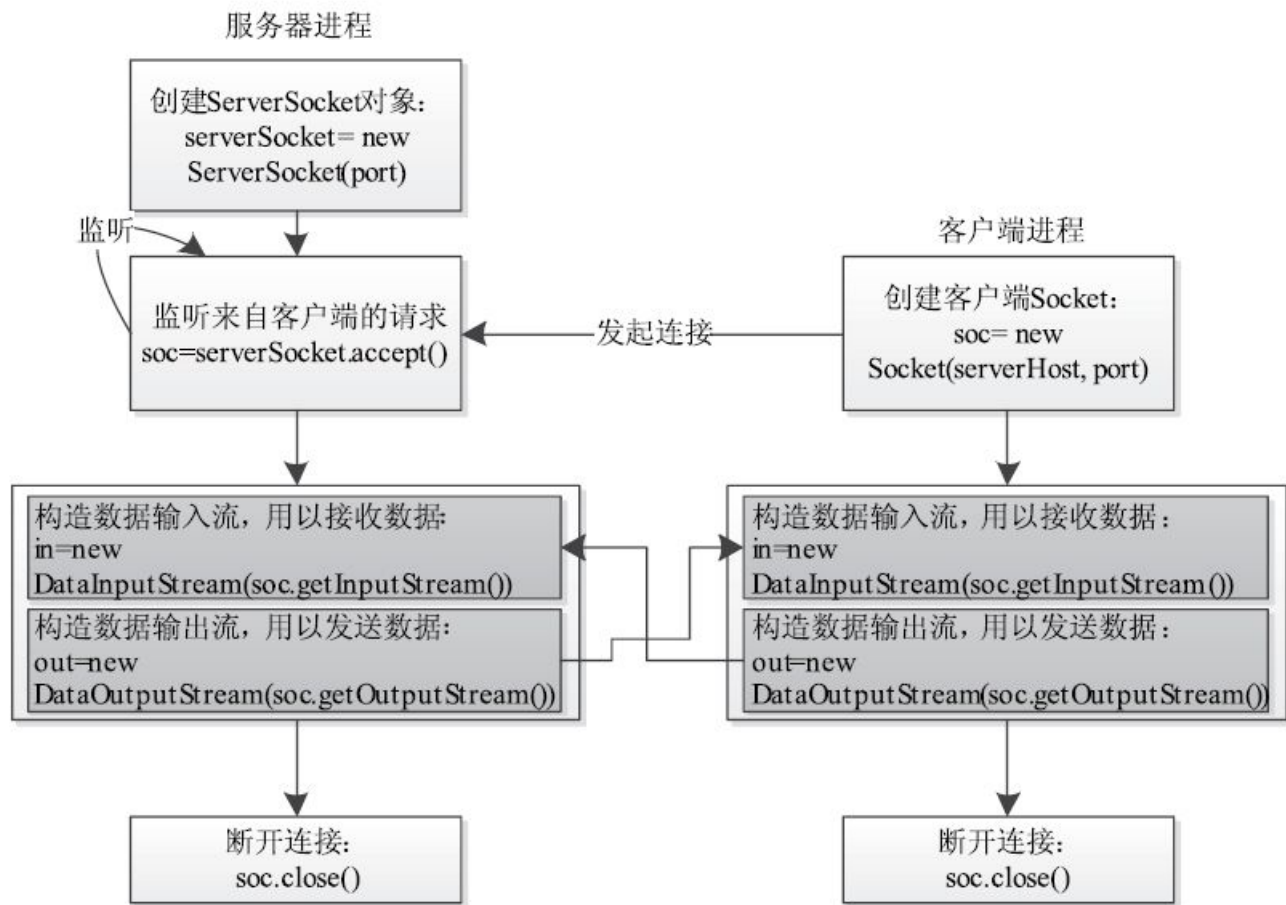


图 4-2 Java客户端/服务器端通信模型

步骤1 创建客户端Socket:

```
Socket soc=new Socket (serverHost, port);
```

其中，serverHost为服务器端的host, port为服务器端的监听端口号。一旦Socket创建成功，则表示客户端连接服务器成功。

步骤2 创建输出、输入流以向服务器端发送数据和从服务器端接收数据:

```
//构造数据输入流, 用以接收数据
DataInputStream in=new DataInputStream (soc.getInputStream());
//构造数据输出流, 用以发送数据
DataOutputStream out=new DataOutputStream (soc.getOutputStream());
.....//应用程序发送和接收数据
```

步骤3 断开连接:

```
soc.close();
```

如图4-2所示，编写一个服务器端程序需要以下4个步骤。

步骤1 创建ServerSocket对象：

```
ServerSocket serverSocket=new ServerSocket (port);
```

其中，**port**为服务器端的监听端口号。当客户端向服务器端建立连接时，需要知道该端口号。创建ServerSocket对象成功后，操作系统将把当前进程注册为服务器进程。

步骤2 监听端口号，等待新连接到达：

```
Socket soc=serverSocket.accept();
```

运行函数accept()后，ServerSocket对象会一直处于监听状态，等待客户端的连接请求。一旦有客户端请求到达，该函数会返回一个Socket对象，该Socket对象与客户端Socket对象形成一条通信链路。

步骤3 创建输出、输入流以向客户端发送数据和从客户端接收数据。此处的程序和客户端的一样，故不再赘述。

步骤4 断开连接。此处的程序和客户端的一样，故不再赘述。

在Client/Server模型中，Server往往需要同时处理大量来自Client的访问请求，因此Server端需采用支持高并发访问的架构。一种简单而又直接的解决方案是“one-thread-per-connection”。这是一种基于阻塞式I/O的多线程模型，如图4-3所示。在该模型中，Server为每个Client连接创建一个处理线程，每个处理线程阻塞式等待可能到达的数据，一旦数据到达，则立即处理请求、返回处理结果并再次进入等待状态。由于每个Client连接有一个单独的处理线程为其服务，因此可保证良好的响应时间。但当系统负载增大（并发请求增多）时，Server端需要的线程数会增加，这将成为系统扩展的瓶颈所在。

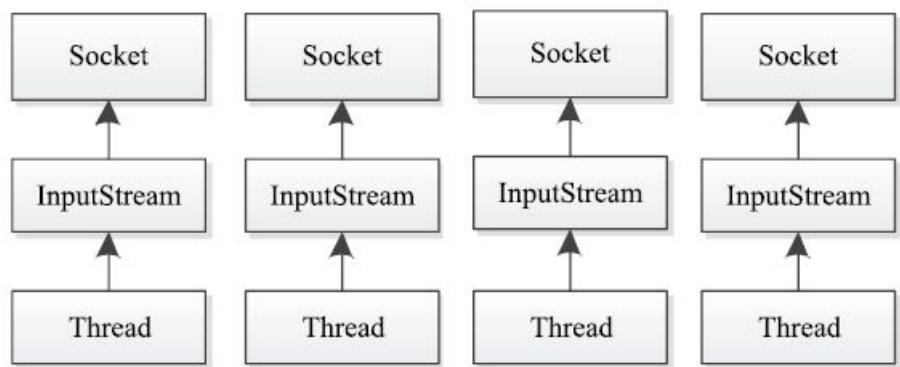


图 4-3 旧I/O模型，多个线程阻塞等待客户端请求

1.简介

自从J2SE 1.4版本以来，JDK发布了全新的I/O类库，简称NIO（New IO）。它不但引入了全新的高效的I/O机制，同时引入了基于Reactor设计模式的多路复用异步模式。NIO的包中主要包含了以下几种抽象数据类型。

□ **Channel（通道）**：NIO把它支持的I/O对象抽象为Channel。它模拟了通信连接，类似于原I/O中的流（Stream），用户可以通过它读取和写入数据。目前已知的实例类有SocketChannel、ServerSocketChannel、DatagramChannel、FileChannel等。

□ **Buffer（缓冲区）**：Buffer是一块连续的内存区域，一般作为Channel收发数据的载体出现。所有数据都通过Buffer对象来处理。用户永远不会将字节直接写入通道中，相反，需将数据写入包含一个或者多个字节的缓冲区；同样，也不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。

□ **Selector（选择器）**：Selector类提供了监控一个或多个通道当前状态的机制。只要Channel向Selector注册了某种特定事件，Selector就会监听这些事件是否会发生，一旦发生某个事件，便会通知对应的Channel。使用选择器，借助单一线程，就可对数量庞大的活动I/O通道实施监控和维护，具体如图4-4所示。

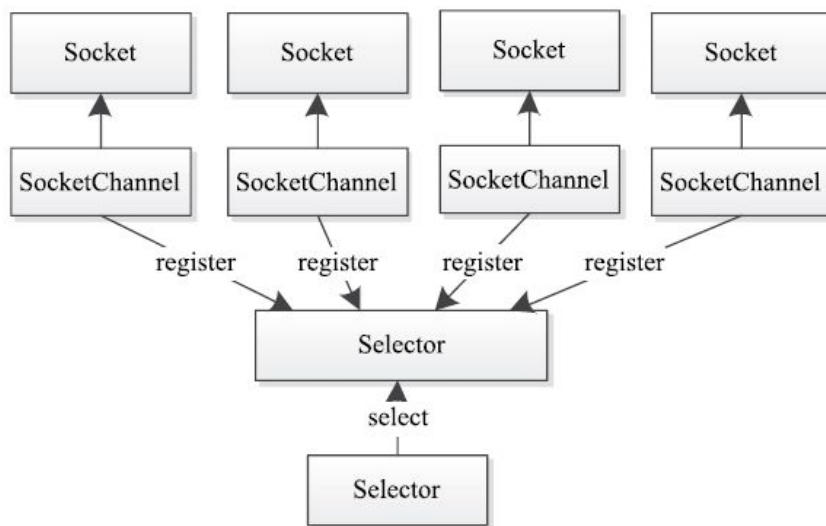


图 4-4 新I/O模型，单个线程阻塞等待客户端请求

2.常用类

(1) Buffer相关类

java.nio包公开了Buffer API，使得Java程序员可以直接控制和运用缓存区。所有缓冲区包含以下3个属性。

□ **capacity**：缓冲区的末位值。它表明了缓冲区最多可以保存多少数据。

□ **limit**：表示缓冲区的当前存放数据的终点。不能对超过limit的区域进行读写数据。

□ **position**：下一个读写单元的位置。每次读写缓冲区时，均会修改该值，为下一次读写数据做准备。

这三个属性的大小关系是 $\text{capacity} \geq \text{limit} \geq \text{position} \geq 0$ 。

如图4-5所示，Buffer有两种不同的工作模式——写模式和读模式。在写模式下，limit与capacity相同，position随着写入数据增加，逐渐增加到limit，因此，0到position之间的数据即为已经写入的数据；在读模式下，limit初始指向position所在位置，position随着数据的读取，逐渐增加到limit，则0到position之间的数据即为已经读取的数据。函数flip()可将写模式转化为读模式，其他常用函数如下。

□ **clear()**：重置Buffer，即将limit设为capacity，而position为0。

- ❑ `hasRemaining()/remaining()`: 分别用于判断Buffer是否有剩余空间和获取Buffer剩余空间，其中剩余空间大小即为`limit-position`。
- ❑ `capacity()/limit()/position()`: 分别用于获取Buffer的`capacity`、`limit`和`position`属性的值。
- ❑ `limit (int newLimit) /position (newPosition)`: 分别用于设置Buffer的`limit`和`position`属性。

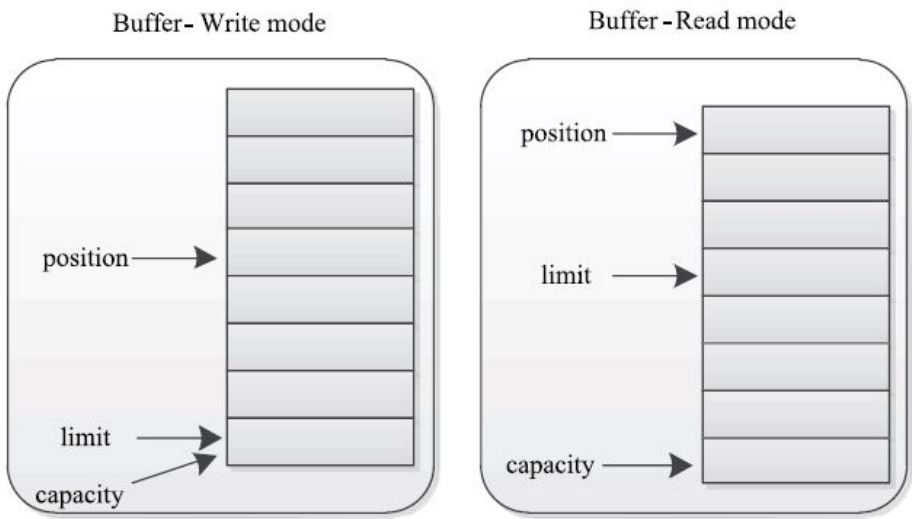


图 4-5 Buffer的写模式和读模式

`java.nio.Buffer`是一个抽象类，不能被实例化。除`boolean`类型外，每种基本类型都有对应的具体的`Buffer`类，其中最基本、最常用的是`ByteBuffer`。它存放的数据单元是字节。它并没有提供直接的构造函数，而是提供了以下两个静态工厂方法：

```
//方法1 创建一个Heap Buffer，其空间分配在JVM的堆上，和其他对象一样，由GC回收
static ByteBuffer allocate (int capacity)
/*方法2 创建一个Direct Buffer，并通过底层的JNI调用C Runtime Time的malloc函数分配空间，可看作“内核空间”。其创建代价比Heap Buffer大，但更高效。*/
static ByteBuffer allocateDirect (int capacity)
```

每个具体类均提供了一系列读写`Buffer`的函数，想进一步了解的读者可查看[JDK Document \[1\]](#)。

(2) Channel相关类

`java.nio`提供了多种`Channel`实现，其中，最常用的是以`SelectableChannel`为基类的通道。`SelectableChannel`是一种支持阻塞I/O和非阻塞I/O的通道，它的主要方法如下：

- ❑ `SelectableChannel configureBlocking (boolean block) throws IOException`。

- 作用：设置当前`SelectableChannel`的阻塞模式。
- 参数含义：`block`表示是否将`SelectableChannel`设置为阻塞模式。
- 返回值：`SelectableChannel`对象本身的引用，相当于“`return this`”。

- ❑ `SelectionKey register (Selector sel, int ops) throws ClosedChannelException`。

- 作用：将当前`Channel`注册到一个`Selector`中。
- 参数含义：`sel`表示要注册的`Selector`；`ops`表示注册事件。
- 返回值：与注册`Channel`关联的`SelectionKey`对象，用于跟踪被注册事件。

`SelectableChannel`的两个子类是`ServerSocketChannel`和`SocketChannel`，它们分别是`ServerSocket`和`Socket`的替代类。

`ServerSocketChannel`主要用于监听TCP连接，它提供了以下3个最常用的方法。

❑ `ServerSocketChannel open()` throws `IOException`: 用于创建 `ServerSocketChannel` 的静态工厂方法。其返回的 `ServerSocketChannel` 对象没有与任何本地端口号绑定，处于阻塞状态。

❑ `SocketChannel accept()` throws `IOException`: 接收来自客户端的连接。当 `ServerSocketChannel` 设置为阻塞模式时，该函数一直会处于阻塞状态，直到有客户端请求到达或者抛出异常。一旦有客户端请求出现，则会返回一个处于阻塞模式的 `SocketChannel`。

❑ `ServerSocket socket()`: 返回一个与 `ServerSocketChannel` 关联的 `ServerSocket` 对象。注意，每个 `ServerSocketChannel` 对象都有一个 `ServerSocket` 对象与之关联。

`SocketChannel` 可看作 `Socket` 的替代类，但功能比 `Socket` 更加强大。同 `ServerSocketChannel` 类似，它提供了静态工厂方法 `open()`（创建对象）和 `socket()` 方法（返回与 `SocketChannel` 关联的 `Socket` 对象）。它的其他常用方法如下。

❑ `boolean connect (SocketAddress remote)` throws `IOException`: 连接 `Channel` 对应的 `Socket`。如果 `SocketChannel` 处于阻塞模式，则直接返回结果；否则，进入阻塞状态，直到连接成功或者抛出异常。

❑ `int read (ByteBuffer dst)` throws `IOException`: 将当前 `Channel` 中的数据读取到 `ByteBuffer` 中。在阻塞和非阻塞模式下，该函数实现方式不同：在非阻塞模式下，遵从能读取多少数据就读取多少数据的原则，总是立即返回结果；而在阻塞模式下，将尝试一直读取数据，直到 `ByteBuffer` 被填满，到达输入流末尾或者抛出异常。该函数的返回值为实际读取的数据字节数。

❑ `int write (ByteBuffer src)` throws `IOException`: 将 `ByteBuffer` 中的数据写入 `Channel` 中。与 `read` 函数类似，在阻塞模式和非阻塞模式下，该函数实现方式不同：在非阻塞模式下，遵从能输出多少数据就输出多少数据的原则，总是立即返回结果；在阻塞模式下，会尝试将所有数据写入 `Channel`，如果底层的网络缓冲区容纳不了这么多字节，则会阻塞至可写入所有数据或者抛出异常。

当调用 `write` 方法将一个 `Heap Buffer` 中的数据写入某个 `Channel`（或者调用 `read` 方法将 `Channel` 中的数据读入一个 `Heap Buffer` 对象）时，Sun Java 底层实现中使用了 `Direct Buffer` 暂时对数据进行缓冲，大体步骤为：JVM 初始创建一个固定大小的 `Direct Buffer`，并将数据写入该 `Buffer`，如果 `Buffer` 大小不够，则再创建一个更大的 `Direct Buffer`，并将之前的 `Direct Buffer` 中的内容复制到新的 `Direct Buffer` 中，依此类推，直到将数据全部写入。很明显，该过程涉及大量内存复制操作，会明显降低性能。此外，由于 `Direct Buffer` 所占内存不会被马上释放，因此会造成内存使用骤升。为解决该问题，可将写入的数据分成固定大小（比如 8KB）的 `chunk`，并以 `chunk` 为单位写入 `Direct Buffer` ^[2]，代码如代码清单 4-2 所示。

代码清单 4-2 将 `ByteBuffer` 中的数据以 `chunk` 为单位写入 `Direct Buffer`

```
.....
int NIO_BUFFER_LIMIT=8*1024; //chunk大小: 8KB
//将Buffer中的数据写入Channel中，其中Channel处于非阻塞模式
int channelWrite (WritableByteChannel channel,
ByteBuffer buffer) throws IOException{
//如果缓冲区中的数据小于8KB，则直接写到Channel中，否则以chunk为单位写入
return (buffer.remaining() <= NIO_BUFFER_LIMIT) ?
channel.write (buffer) : channelIO (null, channel, buffer);
}
private static int channelIO (ReadableByteChannel readCh,
WritableByteChannel writeCh,
ByteBuffer buf) throws IOException{
int originalLimit=buf.limit();
int initialRemaining=buf.remaining();
int ret=0;
while (buf.remaining() > 0) {
try{
int ioSize=Math.min (buf.remaining(), NIO_BUFFER_LIMIT);
buf.limit (buf.position()+ioSize);
ret= (readCh==null) ? writeCh.write (buf) : readCh.read (buf);
//非阻塞模式下，write或者read函数对应的网络缓冲区满后，会直接返回
//返回值为实际写入或者读取的数据
if (ret < ioSize) {
break;
}
}finally{
buf.limit (originalLimit);
}
}
int nBytes=initialRemaining-buf.remaining();
return (nBytes > 0) ? nBytes : ret;
}
```

图 4-6 阐释了按照代码清单 4-2 中的逻辑，`ByteBuffer` 中的数据写入 `Channel` 过程中，其内部各个属性的变化情况。

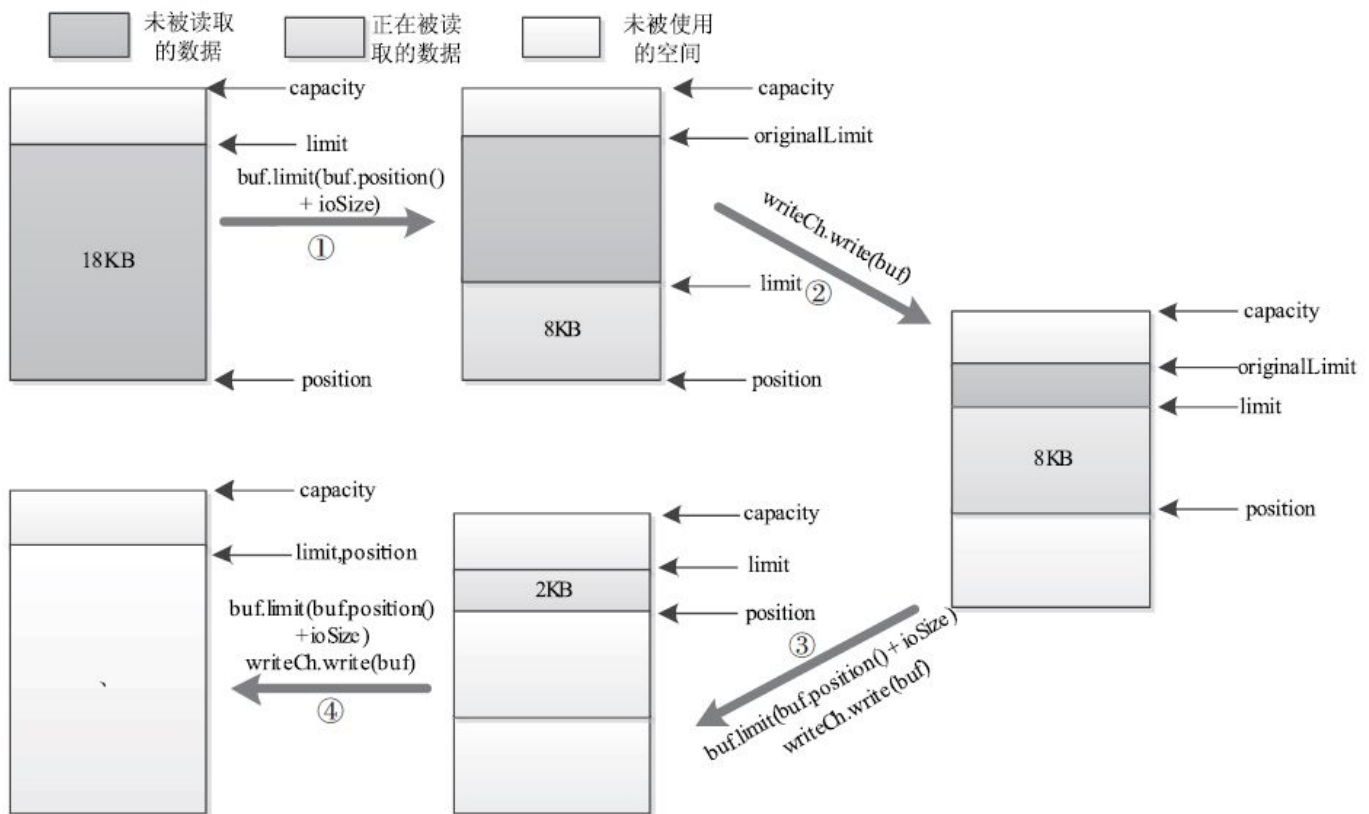


图 4-6 将ByteBuffer中的数据以chunk为单位写入Channel过程

(3) Selector类

Selector可监听ServerSocketChannel和SocketChannel注册的特定事件，一旦某个事件发生，则会通知对应的Channel。SelectableChannel的register()方法负责注册事件，该方法返回一个SelectionKey对象，该对象即为用于跟踪这些注册事件的句柄。

Selector中常用的方法如下。

❑ static Selector open(): 一个静态工厂方法，可用于创建Selector对象。

❑ int select (long timeout): 该方法等待并返回发生的事件。一旦某个注册的事件发生，就会返回对应的SelectionKey的数目，否则，一直处于阻塞状态，直到以下四种情况之一发生：至少一个事件发生；其他线程调用了Selector的wakeup()方法；当前执行select()方法的线程被中断；超出等待时间timeout，如果不设置等待时间，则表示永远不会超时。

❑ set selectedKeys(): Selector捕获的已经发生事件对应的SelectionKey集合。

❑ Selector wakeup(): 立刻唤醒当前处于阻塞状态的Selector。常见应用场景是，线程A调用Selector对象的select()方法，阻塞等待某个注册事件发生，线程B通过调用wakeup()函数可立刻唤醒线程A，使其从select()方法中返回。

(4) SelectionKey类

ServerSocketChannel或SocketChannel通过register()方法向Selector注册事件时，register()方法会创建一个SelectionKey对象，用于跟踪注册事件。在SelectionKey中定义了4种事件，分别用以下4个整型常量表示。

❑ SelectionKey.OP_ACCEPT: 接收（accept）连接就绪事件，表示服务器端接收到了客户端连接。

❑ SelectionKey.OP_CONNECT: 连接就绪事件，表示客户端与服务器端的连接已经建立成功。

❑ SelectionKey.OP_READ: 读就绪事件，表示通道中已经有了可读数据，可执行读操作了。

❑ SelectionKey.OP_WRITE: 写就绪事件，表示可向通道中写入数据了。

通常而言，ServerSocketChannel对象向Selector中注册SelectionKey.OP_ACCEPT事件，而SocketChannel对象向Selector中注册SelectionKey.OP_CONNECT、SelectionKey.OP_READ和SelectionKey.OP_WRITE三种事件。

SelectionKey类中比较重要的方法如下。

❑ Object attach (Object ob)：为当前SelectionKey关联一个Object类型的对象。每个SelectionKey只能关联一个对象。

❑ Object attachment()：获取当前SelectionKey关联的Object对象。

❑ SelectableChannel channel()：返回与当前SelectionKey关联的SelectableChannel对象。

(5) 应用实例——echoServer

echoServer是一个采用NIO编写的服务器。它接收客户端发送过来的字符串，不经任何处理直接再次返回给客户端。

echoServer由五个程序段组成，分别如下。

1) 初始化：

```
public Selector initSelector() throws IOException{
    address=new InetSocketAddress(bindAddress, port);
    //创建一个ServerSocketChannel，并将之设为非阻塞模式
    acceptChannel=ServerSocketChannel.open();
    acceptChannel.configureBlocking(false);
    //将Server Socket绑定到address地址上，并设置监听队列长度
    acceptChannel.socket().bind(address, backlogLength);
    Selector selector=Selector.open();
    //向Selector注册ServerSocketChannel，注册事件为SelectionKey.OP_ACCEPT
    acceptChannel.register(selector, SelectionKey.OP_ACCEPT);
    return selector;
}
```

2) 监听客户端事件，并调用对应事件处理函数：

```
public void run(){
    while (true) {
        SelectionKey key=null;
        try{
            selector.select(); //处于阻塞状态，直到有新事件发生
            Iterator<SelectionKey> iter=selector.selectedKeys().iterator();
            while (iter.hasNext()) {
                key=iter.next();
                iter.remove();
                if (!key.isValid()) {
                    continue;
                }
                if (key.isAcceptable()) {
                    doAccept(key); //新客户端要求建立连接
                } else if (key.isReadable()) {
                    receive(key); //从客户端接收数据
                } else if (key.isWritable()) {
                    send(key); //向客户端发送数据
                }
            }
            key=null;
        } catch (Exception e) {}
    }
}
```

3) 接受新客户端连接请求：

```
private void doAccept (SelectionKey key) throws IOException{
    ServerSocketChannel serverSocketChannel= (ServerSocketChannel) key.channel();
    //接受连接请求，并将之设置为非阻塞模式
    SocketChannel socketChannel=serverSocketChannel.accept();
    socketChannel.configureBlocking(false);
    //将新的SocketChannel注册到Selector中，一旦有需要读或者写的的数据，就会通知相应的程序
    SelectionKey clientKey=
        socketChannel.register(this.selector, SelectionKey.OP_READ);
    ByteBuffer buffer=ByteBuffer.allocate(8192);
    clientKey.attach(buffer); //关联一个缓冲区
}
```

4) 处理读数据请求：

```
private void receive (SelectionKey key) throws IOException{
    SocketChannel socketChannel= (SocketChannel) key.channel();
    ByteBuffer readBuffer= (ByteBuffer) key.attachment();
```



```
socketChannel.read(readBuffer);  
if (numRead>0) {  
    readBuffer.flip();  
    key.interestOps (SelectionKey.OP_WRITE); //切换至OP_WRITE  
}  
}
```

5) 处理写数据请求:

```
private void send (SelectionKey key) throws IOException{  
    SocketChannel socketChannel= (SocketChannel) key.channel();  
    ByteBuffer writeBuffer= (ByteBuffer) key.attachment();  
    socketChannel.write (writeBuffer);  
    if (writeBuffer.remaining() == 0) { //写完成后, 切换至SelectionKey.OP_READ  
        writeBuffer.clear();  
        key.interestOps (SelectionKey.OP_READ);  
    }  
}
```

[1] <http://docs.oracle.com/javase/1.4.2/docs/guide/nio/>

[2] 可参考<https://issues.apache.org/jira/browse/HADOOP-4797>下的相关文档。

4.3 Hadoop RPC基本框架分析

4.3.1 RPC基本概念

RPC是一种通过网络从远程计算机上请求服务，但不需要了解底层网络技术的协议。RPC协议假定某些传输协议已经存在，如TCP或UDP等，并通过这些传输协议为通信程序之间传递访问请求或者应答信息。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发分布式应用程序更加容易 [1]。

RPC通常采用客户机/服务器模型。请求程序是一个客户机，而服务提供程序则是一个服务器。一个典型的RPC框架主要包括以下几个部分。

□通信模块：两个相互协作的通信模块实现请求-应答协议。它们在客户机和服务器之间传递请求和应答消息，一般不会对数据包进行任何处理。

请求-应答协议的实现方式有两种，分别是同步方式和异步方式。如图4-7所示，同步模式下客户端程序一直阻塞到服务器端发送的应答请求到达本地；而异步模式则不同，客户端将请求发送到服务器端后，不必等待应答返回，可以做其他事情，待服务器端处理完请求后，主动通知客户端。在高并发应用场景中，一般采用异步模式以降低访问延迟和提高带宽利用率。

□Stub程序：客户端和服务端均包含Stub程序，可将之看作代理程序。它使得远程函数调用表现的跟本地调用一样，对用户程序完全透明。在客户端，它表现的就像一个本地程序，但不直接执行本地调用，而是将请求信息通过网络模块发送给服务器端。此外，当服务器端发送应答后，它会解码对应结果。在服务器端，Stub程序依次进行以下处理：解码请求消息中的参数、调用相应的服务过程和编码应答结果的返回值。

□调度程序：调度程序接收来自通信模块的请求消息，并根据其中的标识选择一个Stub程序处理。通常客户端并发请求量比较大时，会采用线程池提高处理效率。

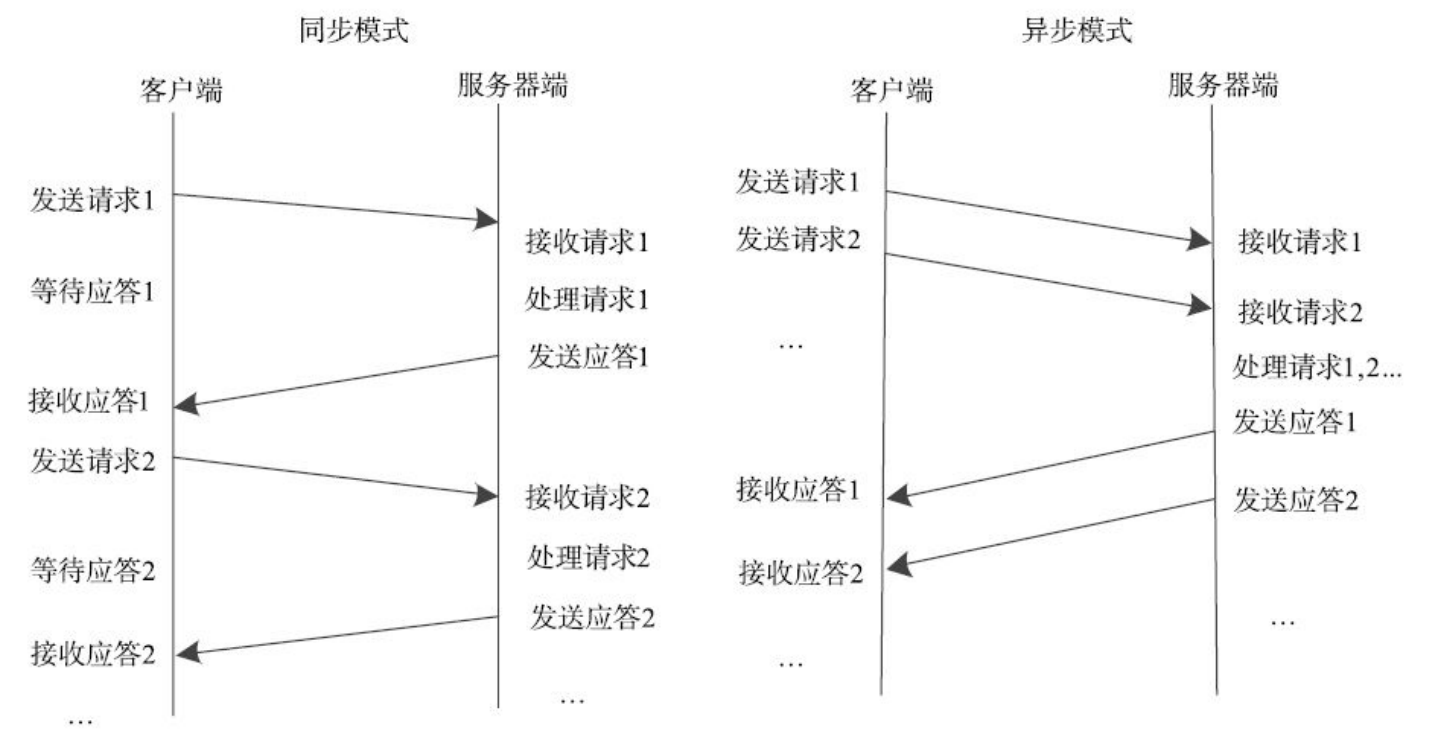


图 4-7 同步模式与异步模式对比

□客户程序/服务过程：请求的发出者和请求的处理者。如果是单机环境，客户程序可直接通过函数调用访问服务过程，但在分布式环境下，需要考虑网络通信，这不得不增加通信模块和Stub程序（保证函数调用的透明性）。

通常而言，一个RPC请求从发送到获取处理结果，所经历的步骤如下（见图4-8）：

步骤1 客户程序以本地方式调用系统产生的Stub程序；

步骤2 该Stub程序将函数调用信息按照网络通信模块的要求封装成消息包，并交给通信模块发送到远程服务器端；

步骤3 远程服务器端接收此消息后，将此消息发送给相应的Stub程序；

步骤4 Stub程序拆封消息，形成被调过程要求的形式，并调用对应的函数；

步骤5 被调函数按照所获参数执行，并将结果返回给Stub程序；

步骤6 Stub程序将此结果封装成消息，通过网络通信模块逐级地传送给客户程序。

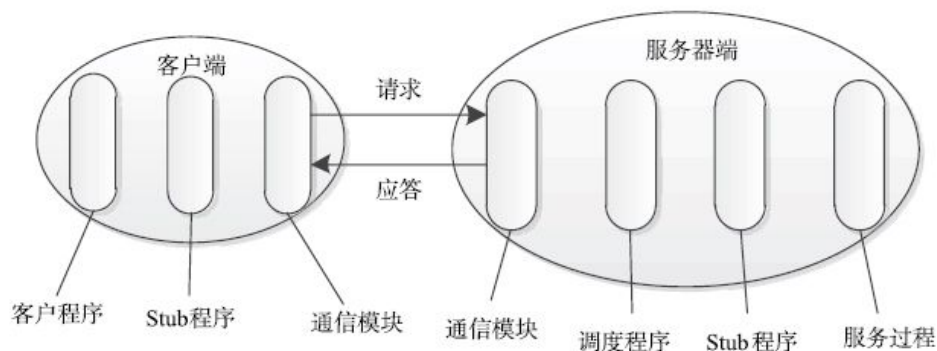


图 4-8 RPC通用架构

[1] http://en.wikipedia.org/wiki/Remote_Procedure_call

4.3.2 Hadoop RPC基本框架

1.Hadoop RPC使用

在正式介绍Hadoop RPC基本框架之前，先介绍怎么样使用它。Hadoop RPC主要对外提供了两种接口。

□`public static VersionedProtocol getProxy/waitForProxy()`: 构造一个客户端代理对象（该对象实现了某个协议），用于向服务器端发送RPC请求。

□`public static Server getServer()`: 为某个协议（实际上是Java接口）实例构造一个服务器对象，用于处理客户端发送的请求。

通常而言，Hadoop RPC使用方法可分为以下几个步骤。

步骤1 定义RPC协议。RPC协议是客户端和服务端之间的通信接口，它定义了服务器端对外提供的服务接口。如以下代码所示，我们定义了一个`ClientProtocol`通信接口，它声明了两个方法：`echo()`和`add()`。需要注意的是，Hadoop中所有自定义RPC接口都需要继承`VersionedProtocol`接口，它描述了协议的版本信息。

```
interface ClientProtocol extends org.apache.hadoop.ipc.VersionedProtocol{
//版本号。默认情况下，不同版本号的RPC Client和Server之间不能相互通信
public static final long versionID=1L;
String echo (String value) throws IOException;
int add (int v1, int v2) throws IOException; }
```

步骤2 实现RPC协议。Hadoop RPC协议通常是一个Java接口，用户需要实现该接口。如以下代码所示，对`ClientProtocol`接口进行简单的实现：

```
public static class ClientProtocolImpl implements ClientProtocol{
public long getProtocolVersion (String protocol, long clientVersion) {
return ClientProtocol.versionID;
}
public String echo (String value) throws IOException{
return value;
}
public int add (int v1, int v2) throws IOException{
return v1+v2;
}
}
```

步骤3 构造并启动RPC Server。直接使用静态方法`getServer()`构造一个RPC Server，并调用函数`start()`启动该Server:

```
server=RPC.getServer (new ClientProtocolImpl(), serverHost, serverPort,
numHandlers, false, conf);
server.start();
```

其中，`serverHost`和`serverPort`分别表示服务器的host和监听端口号，而`numHandlers`表示服务器端处理请求的线程数目。到此为止，服务器处理监听状态，等待客户端请求到达。

步骤4 构造RPC Client，并发送RPC请求。使用静态方法`getProxy()`构造客户端代理对象，直接通过代理对象调用远程端的方法，具体如下所示：

```
proxy=(ClientProtocol) RPC.getProxy (
ClientProtocol.class, ClientProtocol.versionID, addr, conf);
int result=proxy.add (5, 6);
String echoResult=proxy.echo ("result");
```

经过以上四步，我们便利用Hadoop RPC搭建了一个非常高效的客户机/服务器网络模型。接下来，我们将深入Hadoop RPC内部，剖析它的设计原理及设计技巧。

Hadoop RPC主要由三个大类组成，分别是RPC、Client和Server，分别对应对外编程接口、客户端实现和服务端实现。

2.ipc.RPC类分析

RPC类实际上是对底层客户机/服务器网络模型的封装，以便为程序员提供一套更方便简洁的编程接口。

如图4-9所示，RPC类自定义了一个内部类RPC.Server。它继承Server抽象类，并利用Java反射机制实现了call接口（Server抽象类中并未给出该接口的实现），即根据客户端请求中的调用方法名称和对应参数完成方法调用。RPC类包含一个ClientCache类型的成员变量，它根据用户提供的SocketFactory缓存Client对象，以达到重用Client对象的目的。

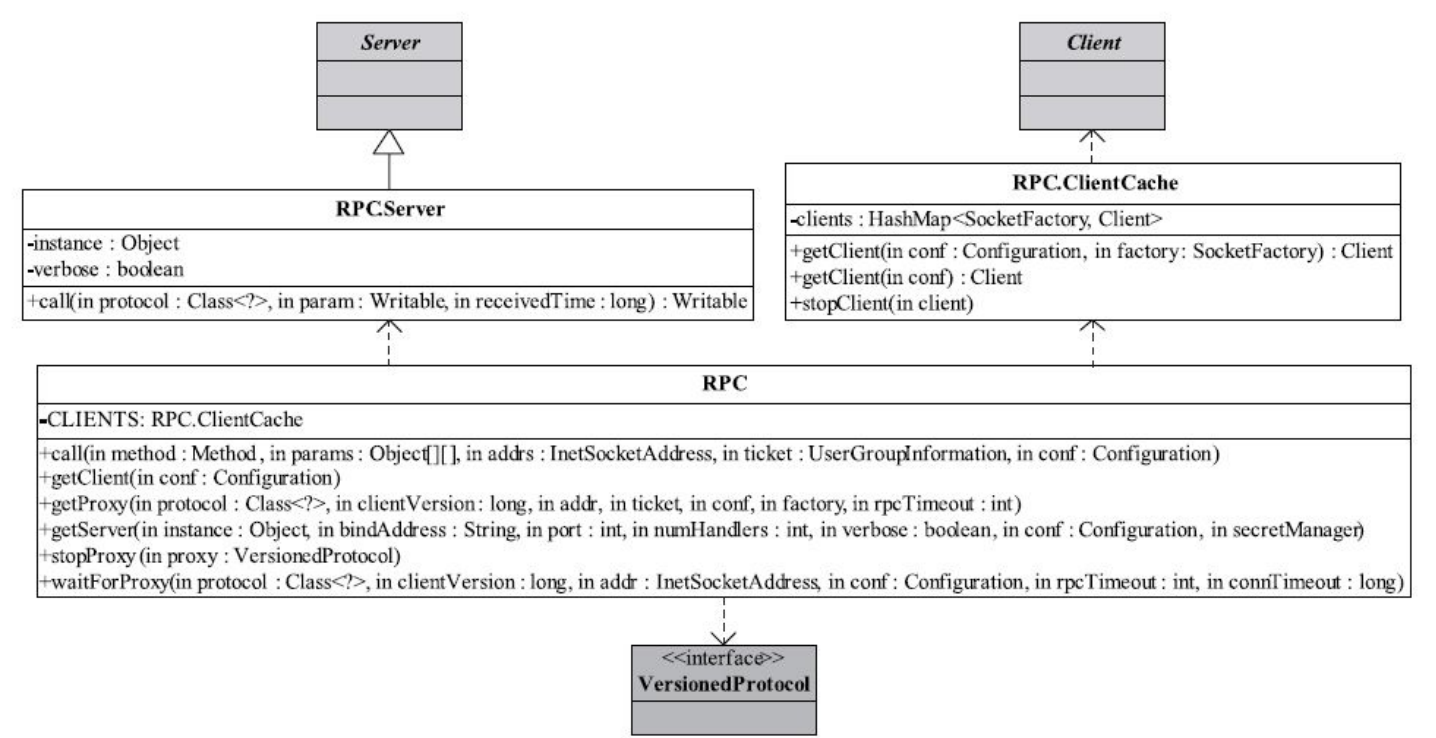


图 4-9 Hadoop RPC的主要类关系图

Hadoop RPC使用了Java动态代理完成对远程方法的调用。在4.2.1小节中，我们介绍了Java动态代理机制：用户只需实现java.lang.reflect.InvocationHandler接口，并按照自己的需求实现invoke方法即可完成动态代理类对象上的方法调用；我们还在代码中给出了一个本地动态代理实例。但对于Hadoop RPC，函数调用由客户端发出，并在服务器端执行并返回，因此不能像4.2.1节的本地动态代理实例代码一样直接在invoke方法中本地调用相关函数，它的做法是，在invoke方法中，将函数调用信息（函数名、函数参数列表等）打包成可序列化的Invocation对象，并通过网络发送给服务器端，服务器端收到该调用信息后，解析出函数名和函数参数列表等信息，利用Java反射机制完成函数调用，期间涉及的类关系如图4-10所示。

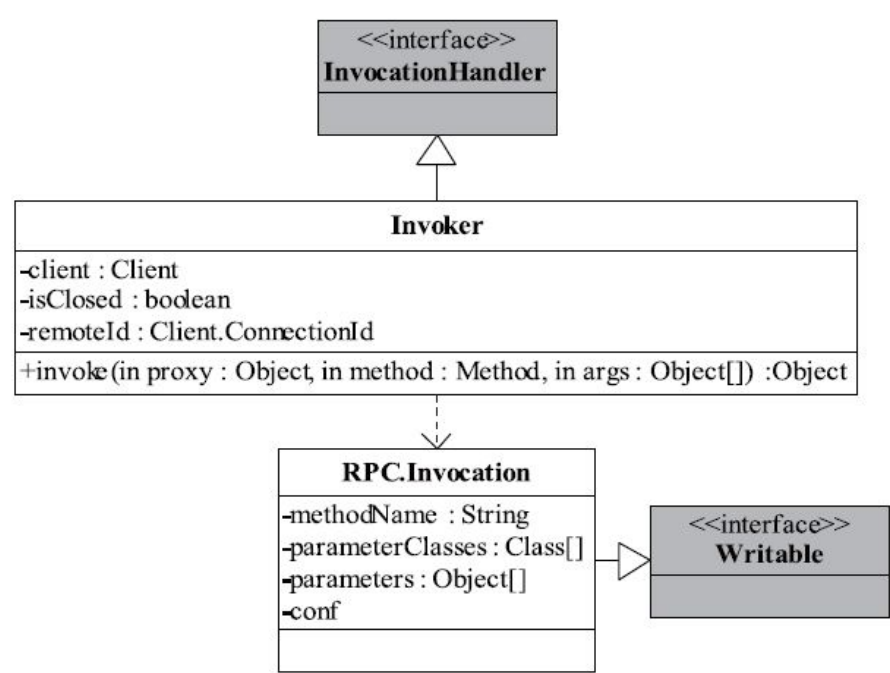


图 4-10 Hadoop RPC中服务器端动态代理实现类图

3.ipc.Client类分析

Client主要完成的功能是发送远程过程调用信息并接收执行结果。它涉及的类关系如图4-11所示。**Client**类对外提供了两种接口，一种用于执行单个远程调用。另外一种用于执行批量远程调用。它们的声明如下所示：

```
public Writable call (Writable param, ConnectionId remoteId)
throws InterruptedException, IOException;
public Writable[] call (Writable[]params, InetSocketAddress[]addresses,
Class<?>protocol, UserGroupInformation ticket, Configuration conf)
throws IOException, InterruptedException;
```

Client内部有两个重要的内部类，分别是**Call**和**Connection**。

❑ **Call**类：该类封装了一个RPC请求，它包含五个成员变量，分别是唯一标识**id**、函数调用信息**param**、函数执行返回值**value**、出错或者异常信息**error**和执行完成标识符**done**。由于Hadoop RPC Server采用了异步方式处理客户端请求，这使得远程过程调用的发生顺序与结果返回顺序无直接关系，而**Client**端正是通过**id**识别不同的函数调用。当客户端向服务器端发送请求时，只需填充**id**和**param**两个变量，而剩下的三个变量：**value**、**error**和**done**，则由服务器端根据函数执行情况填充。

❑ **Connection**类：**Client**与每个**Server**之间维护一个通信连接。该连接相关的基本信息及操作被封装到**Connection**类中。其中，基本信息主要包括：通信连接唯一标识（**remoteId**），与**Server**端通信的**Socket**（**socket**），网络输入数据流（**in**），网络输出数据流（**out**），保存RPC请求的哈希表（**calls**）等；操作则包括：

❑ **addCall**——将一个**Call**对象添加到哈希表中；

❑ **sendParam**——向服务器端发送RPC请求；

❑ **receiveResponse**——从服务器端接收已经处理完成的RPC请求；

❑ **run**——**Connetion**是一个线程类，它的**run**方法调用了**receiveResponse**方法，会一直等待接收RPC返回结果。

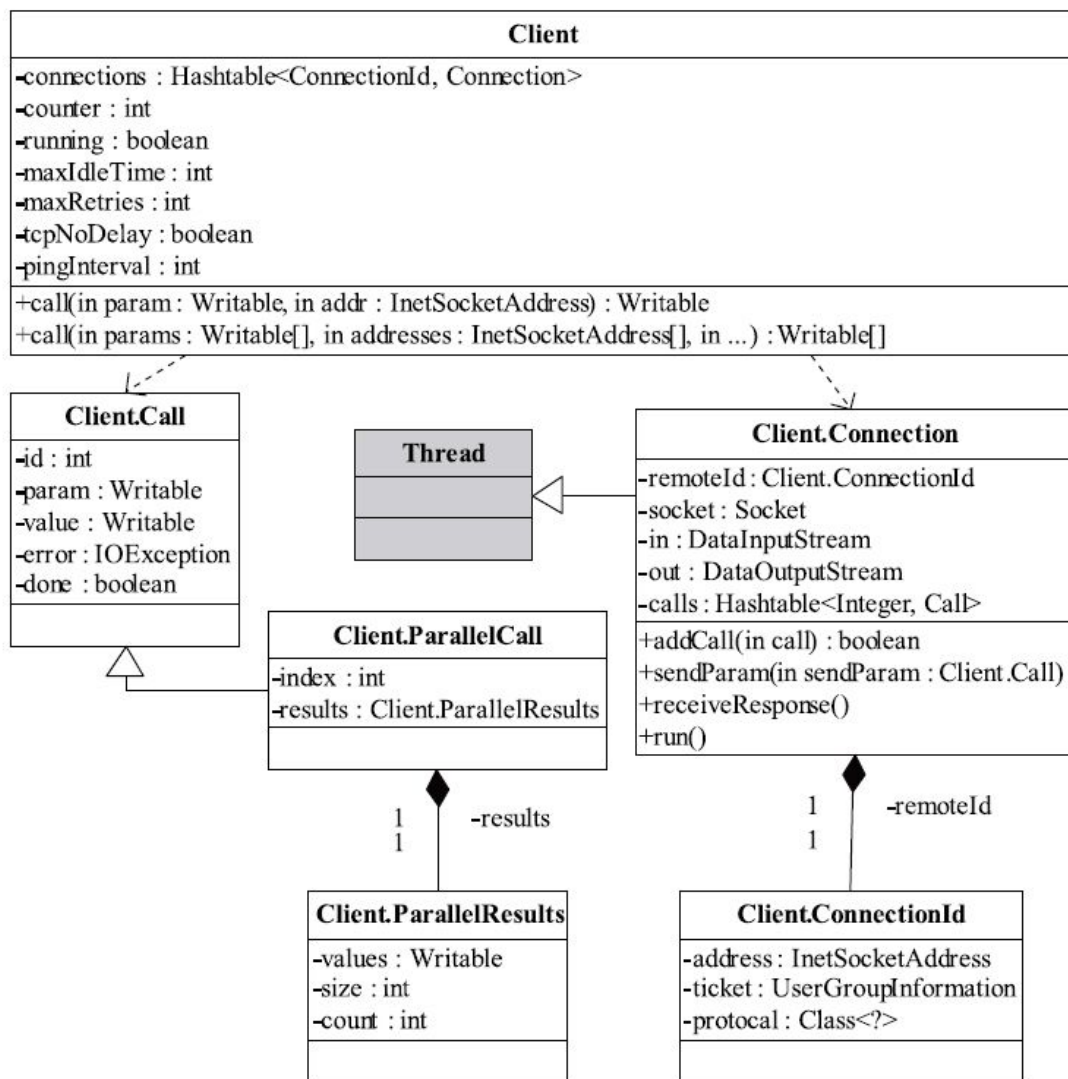


图 4-11 Client类图

当调用call函数执行某个远程方法时，Client端需要进行如图4-12所示的几个步骤：

- 步骤1 创建一个Connection对象，并将远程方法调用信息封装成Call对象，放到Connection对象中的哈希表calls中；
- 步骤2 调用Connection类中的sendParam()方法将当前Call对象发送给Server端；
- 步骤3 Server端处理完RPC请求后，将结果通过网络返回给Client端，Client端通过receiveResponse()函数获取结果；
- 步骤4 Client端检查结果处理状态（成功还是失败），并将对应的Call对象从哈希表中删除。

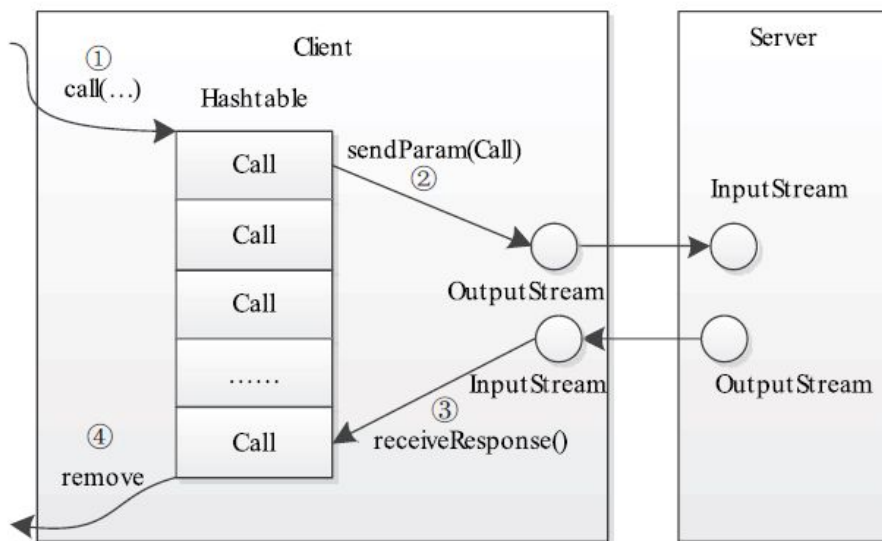


图 4-12 Hadoop RPC Client处理流程

4.ipc.Server类分析

Hadoop采用了Master/Slave结构。其中，Master是整个系统的单点，如NameNode或JobTracker，这是制约系统性能和可扩展性的最关键因素之一，而Master通过ipc.Server接收并处理所有Slave发送的请求，这就要求ipc.Server将高并发和可扩展性作为设计目标。为此，ipc.Server采用了很多具有提高并发处理能力的技术，主要包括线程池、事件驱动和Reactor设计模式等。这些技术均采用了JDK自带的库实现。这里重点分析它是如何利用Reactor设计模式提高整体性能的。

Reactor是并发编程中的一种基于事件驱动的设计模式。它具有以下两个特点：①通过派发/分离I/O操作事件提高系统的并发性能；②提供了粗粒度的并发控制，使用单线程实现，避免了复杂的同步处理。一个典型的Reactor实现原理图如图4-13所示。

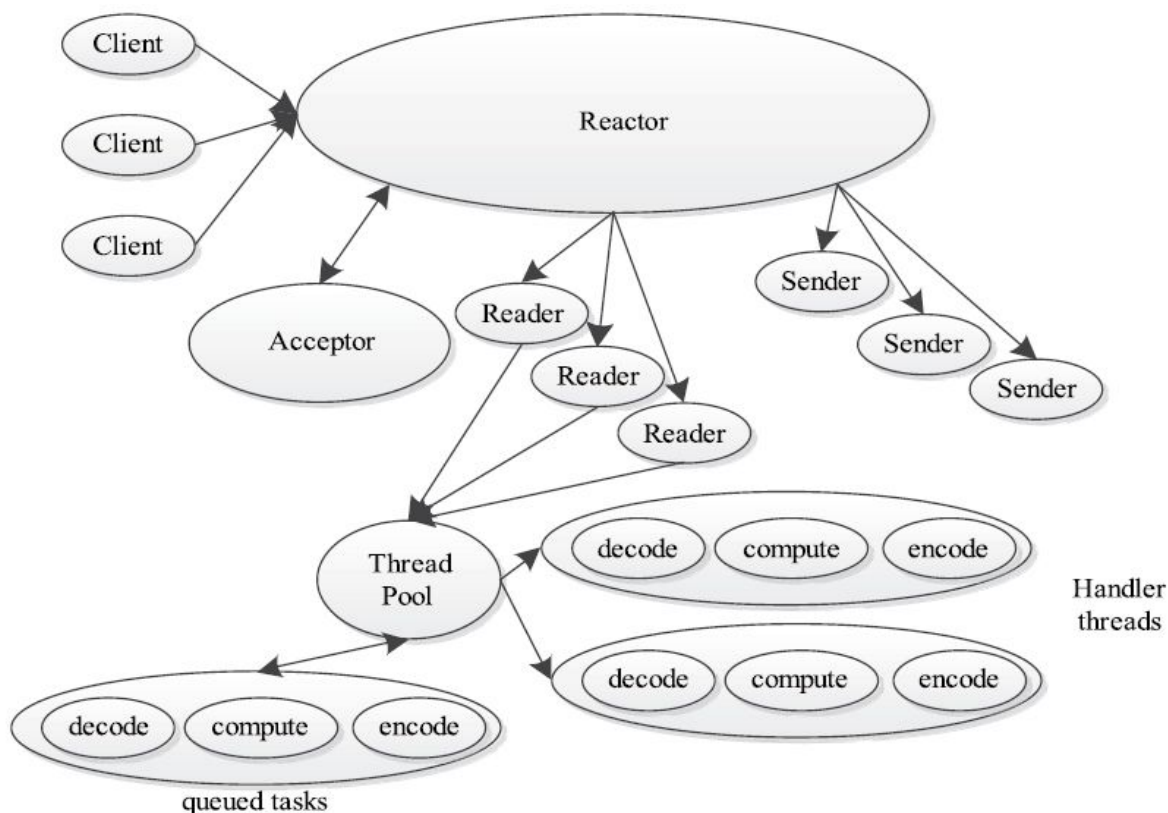


图 4-13 Reactor模式工作原理图

一个典型的Reactor模式中主要包括以下几个角色。

□ Reactor: IO事件的派发者。

□ **Acceptor**: 接受来自Client的连接，建立与Client对应的Handler，并向Reactor注册此Handler。

□ **Handler**: 与一个Client通信的实体，并按一定的过程实现业务的处理。Handler内部往往会有更进一步的层次划分，用来抽象诸如read, decode, compute, encode和send等的过程。在Reactor模式中，业务逻辑被分散的IO事件所打破，所以Handler需要有适当的机制在所需的信息还不全（读到一半）的时候保存上下文，并在下一次IO事件到来的时候（另一半可读了）能继续上次中断的处理。

□ **Reader/Sender**: 为了加速处理速度，Reactor模式往往构建一个存放数据处理线程的线程池，这样，数据读出后，立即扔到线程池中等待后续处理即可。为此，Reactor模式一般分离Handler中的读和写两个过程，分别注册成单独的读事件和写事件，并由对应的Reader和Sender线程处理。

ipc.Server实际上实现了一个典型的Reactor设计模式，其整体架构与上述完全一致。读者一旦了解典型的Reactor架构，便可很容易地学习ipc.Server的设计思路及实现。接下来，我们分析ipc.Server的实现细节。

前面提到，ipc.Server的主要功能是接收来自客户端的RPC请求，经过调用相应的函数获取结果后，返回给对应的客户端。为此，ipc.Server被划分成三个阶段：接收请求，处理请求和返回结果。如图4-14所示，各阶段实现细节如下：

(1) 接收请求

该阶段的主要任务是接收来自各个客户端的RPC请求，并将它们封装成固定的格式（Call类）放到一个共享队列（callQueue）中，以便进行后续处理。该阶段内部又分为两个子阶段：建立连接和接收请求，分别由两种线程完成：Listener和Reader。

整个Server只有一个Listener线程，统一负责监听来自客户端的连接请求。一旦有新的请求到达，它会采用轮询的方式从线程池中选一个Reader线程进行处理。而Reader线程可同时存在多个，它们分别负责接收一部分客户端连接的RPC请求。至于每个Reader线程负责哪些客户端连接，完全由Listener决定。当前Listener只是采用了简单的轮询分配机制。

Listener和Reader线程内部各自包含一个Selector对象，分别用于监听SelectionKey.OP_ACCEPT和SelectionKey.OP_READ事件。对于Listener线程，主循环的实现体是监听是否有新的连接请求到达，并采用轮询策略选择一个Reader线程处理新连接；对于Reader线程，主循环的实现体是监听（它负责的那部分）客户端连接中是否有新的RPC请求到达，并将新的RPC请求封装成Call对象，放到共享队列callQueue中。

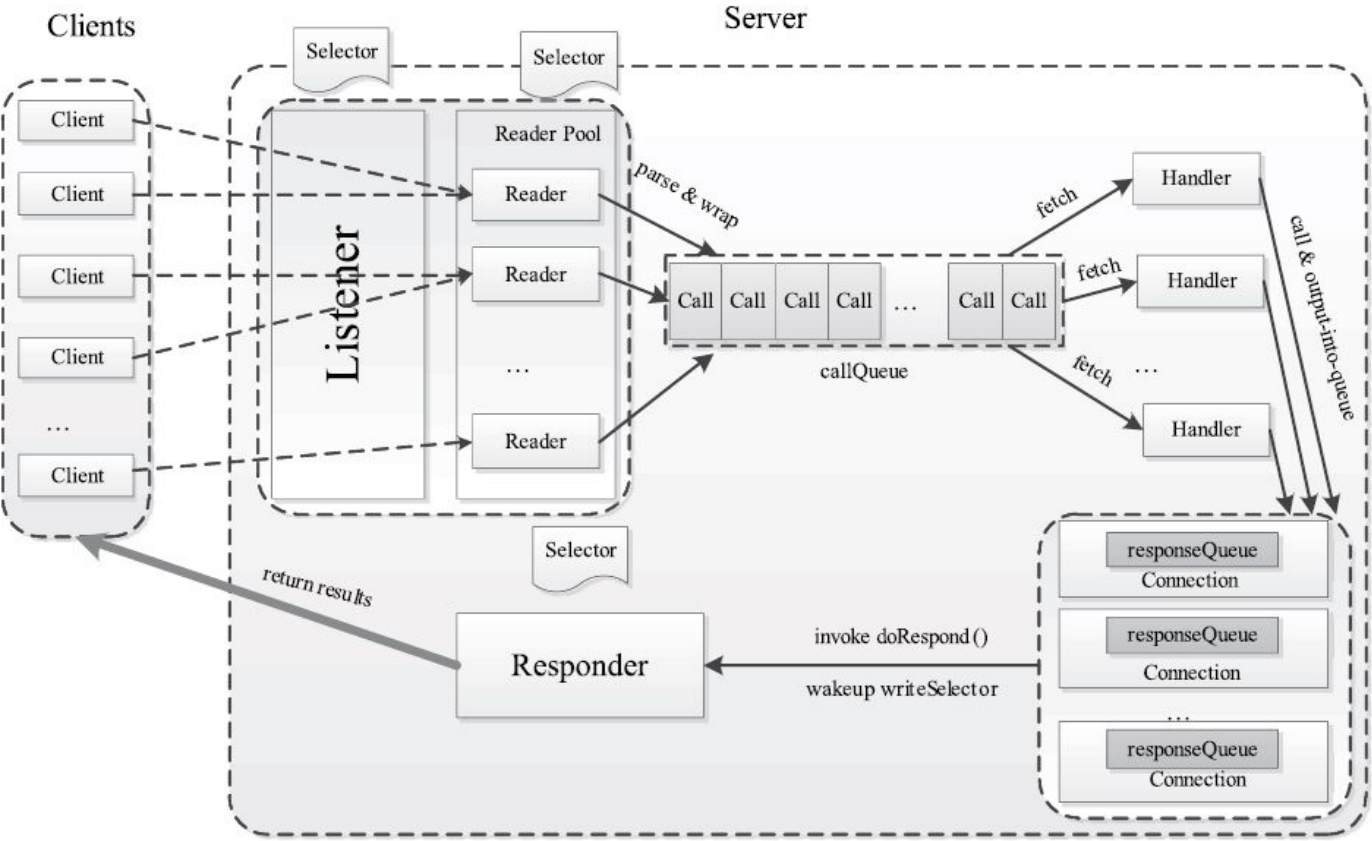


图 4-14 Hadoop RPC Server处理流程

（2）处理请求

该阶段的主要任务是从共享队列callQueue中获取Call对象，执行对应的函数调用，并将结果返回给客户端，这全部由Handler线程完成。

Server端可同时存在多个Handler线程。它们并行从共享队列中读取Call对象，经执行对应的函数调用后，将尝试着直接将结果返回给对应的客户端。但考虑到某些函数调用返回的结果很大或者网络速度过慢，可能难以将结果一次性发送到客户端，此时Handler将尝试着将后续发送任务交给Responder线程。

（3）返回结果

前面提到，每个Handler线程执行完函数调用后，会尝试着将执行结果返回给客户端，但对于特殊情况，比如函数调用返回的结果过大或者网络异常情况（网速过慢），会将发送任务交给Responder线程。

Server端仅存在一个Responder线程。它的内部包含一个Selector对象，用于监听SelectionKey.OP_WRITE事件。当Handler没能够将结果一次性发送到客户端时，会向该Selector对象注册SelectionKey.OP_WRITE事件，进而由Responder线程采用异步方式继续发送未发送完成的结果。

5.Hadoop RPC参数调优

Hadoop RPC对外提供了一些可配置参数，以便于用户根据业务需求和硬件环境对其进行调优，主要的配置参数如下。

❑ Reader线程数目：由参数ipc.server.read.threadpool.size配置，默认是1。也就是说，默认情况下，一个RPC Server只包含一个Reader线程。

❑ 每个Handler线程对应的最大Call数目：由参数ipc.server.handler.queue.size指定，默认是100。也就是说，默认情况下，每个Handler线程对应的Call队列长度为100。比如，如果Handler数目为10，则整个Call队列（共享队列callQueue）最大长度为：100×10=1 000。

❑ Handler线程数目：在Hadoop中，JobTracker和NameNode分别是MapReduce和HDFS两个子系统下的RPC Server，其对应的Handler数目分别由参数mapred.job.tracker.handler.count和dfs.namenode.service.handler.count指定，默认值均为10。当集群规模较大时，这两个参数值会大大影响系统性能。

❑ 客户端最大重试次数：在分布式环境下，因网络故障或者其他原因迫使客户端重试连接是很常见的，但尝试次数过多可能不利于对实时性要求较高的应用。客户端最大重试次数由参数ipc.client.connect.max.retries指定，默认值为10，也就是会连续尝试10次（每两次之间相隔1秒钟）。

4.3.3 集成其他开源RPC框架

当前存在非常多的开源RPC框架，比较有名的有Thrift [1], Protocol Buffers [2] 和Avro [3]。与Hadoop RPC一样，它们均由两部分组成：对象序列化和远程过程调用。相比于Hadoop RPC，它们有以下几个特点。

□ 跨语言特性：前面提到，RPC框架实际上是客户机/服务器模型的一个应用实例。对于Hadoop RPC而言，由于Hadoop采用Java语言编写，因而其RPC客户端和服务端仅支持Java语言；但对于更通用的RPC框架，如Thrift或者Protocol Buffers等，其客户端和服务端可采用任何语言编写，如Java, C++, Python等，这给用户编程带来极大的方便。

□ 引入IDL：开源RPC框架均提供了一套接口描述语言（Interface Description Language, IDL）。它提供一套通用的数据类型，并以这些数据类型来定义更为复杂的数据类型和对外服务接口。一旦用户按照IDL定义的语法编写完接口文件后，即可根据实际需要生成特定的编程语言（如Java, C++, Python等）的客户端和服务端代码。

□ 协议兼容性：开源RPC框架在设计上均考虑到了协议兼容性问题，即当协议格式发生改变时，比如某个类需要添加或者删除一个成员变量（字段）后，旧版本代码仍然能识别新格式的数据，也就是说，具有向后兼容性。

随着Hadoop版本的不断演化，研究人员发现Hadoop RPC在跨语言支持和协议兼容性两个方面存在不足，具体表现为：

1) 从长远发展看，Hadoop RPC应允许某些协议的客户端或者服务端采用其他语言实现，比如用户希望直接使用C/C++语言读写HDFS中的文件，这就需要有C/C++语言的HDFS客户端。

2) 当前Hadoop版本较多，而不同版本之间不能通信。比如，0.20.2版本的JobTracker不能与0.21.0版本中的TaskTracker通信，如果用户企图这样做，会抛出VersionMismatch异常。

为了解决以上几个问题，从0.21.0版本开始，Hadoop尝试着将RPC中的序列化部分剥离开，以便将现有的开源RPC框架集成进来。改进之后，Hadoop RPC的类关系如图4-15所示。RPC类变成了一个工厂，它将具体的RPC实现授权给RpcEngine实现类，而现有的开源RPC只要实现RpcEngine接口，便可以集成到Hadoop RPC中。在该图中，WritableRpcEngine是采用Hadoop自带的序列化框架实现的RPC，而AvroRpcEngine [4] 和ProtobufRpcEngine [5] 分别是开源RPC框架Avro和Protocol Buffers对应的RpcEngine接口实现。用户可通过配置参数rpc.engine.{protocol}以指定协议{protocol}采用的RPC框架。需要注意的是，当前实现中，Hadoop RPC只采用这些开源框架的序列化机制，底层的函数调用机制仍采用Hadoop自带的。

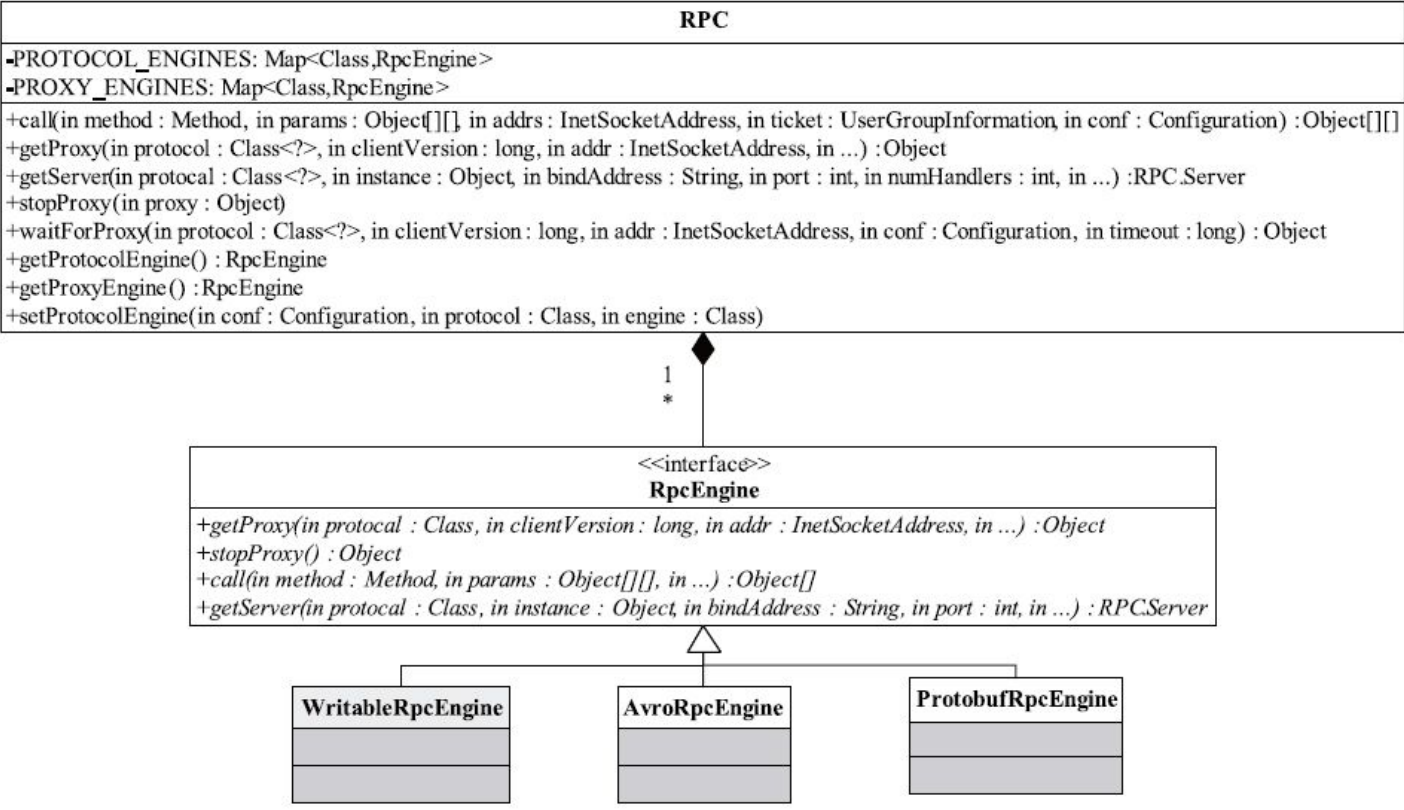


图 4-15 Hadoop RPC集成多种开源RPC框架

在下一代的Hadoop（参见第12章）中，已将Protocol Buffers作为默认的序列化机制 [6]（而不是Hadoop自带的Writable），这带来的好处主要表现在以下几个方面。

（1）继承了Protocol Buffers的优势

Protocol Buffers已在实践中证明了其高效性，可扩展性，紧凑性和跨语言特性。首先，它允许在保持向后兼容性的前提下修改协议，比如为某个定义好的数据格式添加一个新的字段；其次，它支持多种语言，进而方便用户为某些服务（比如HDFS的NameNode）编写非Java客户端；此外，实验表明Protocol Buffers比Hadoop自带的Writable在性能方面有很大提升。

（2）支持升级回滚

Hadoop 2. 0.0及其以上版本中已经将NameNode HA方案合并进来。在该方案中，NameNode有两种角色：Active和Standby。其中，Active NameNode是当前对外提供服务，而Standby NameNode则能够在Active NameNode出现故障时接替它。采用Protocol Buffers序列化机制后，管理员能够在不停止NameNode对外服务的前提下，通过主备NameNode之间的切换，依次对主备NameNode进行在线升级（不用考虑版本和协议兼容性问题）。

[1] <http://thrift.apache.org/>
[2] <http://code.google.com/p/protobuf/>
[3] <http://avro.apache.org/>
[4] AvroRpcEngine从Hadoop 0.21.0版本开始出现。
[5] ProtobufRpcEngine从Hadoop 2.0-apha版本开始出现。
[6] <https://issues.apache.org/jira/browse/HADOOP-7347>

4.4 MapReduce通信协议分析

本书重点介绍MapReduce，因此对Hadoop RPC上层系统的分析也只限于MapReduce分布式计算框架。在Hadoop MapReduce中，不同组件之间的通信协议均是基于RPC的。它们就像系统的“骨架”，支撑起整个MapReduce系统。本节我们将详细介绍Hadoop MapReduce中所有基于RPC的通信协议。

4.4.1 MapReduce通信协议概述

在Hadoop 1.0.0版本中，MapReduce框架中共有6个主要的通信协议，具体如图4-16所示。其中，直接面向Client（用户）的通信协议共有4个。

- ❑ JobSubmissionProtocol^[1]：Client（一般为普通用户）与JobTracker之间的通信协议。用户通过该协议提交作业，查看作业运行情况等。
- ❑ RefreshUserMappingsProtocol：Client（一般为管理员）通过该协议更新用户-用户组映射关系。
- ❑ RefreshAuthorizationPolicyProtocol：Client（一般为管理员）通过该协议更新MapReduce服务级别访问控制列表。
- ❑ AdminOperationsProtocol：Client（一般为管理员）通过该协议更新队列（存在于JobTracker或者Scheduler中）访问控制列表和节点列表。

在Hadoop线上环境中，考虑到安全因素，通常将JobSubmissionProtocol使用权限授予普通用户，而其他三个通信协议的权限授予管理员。

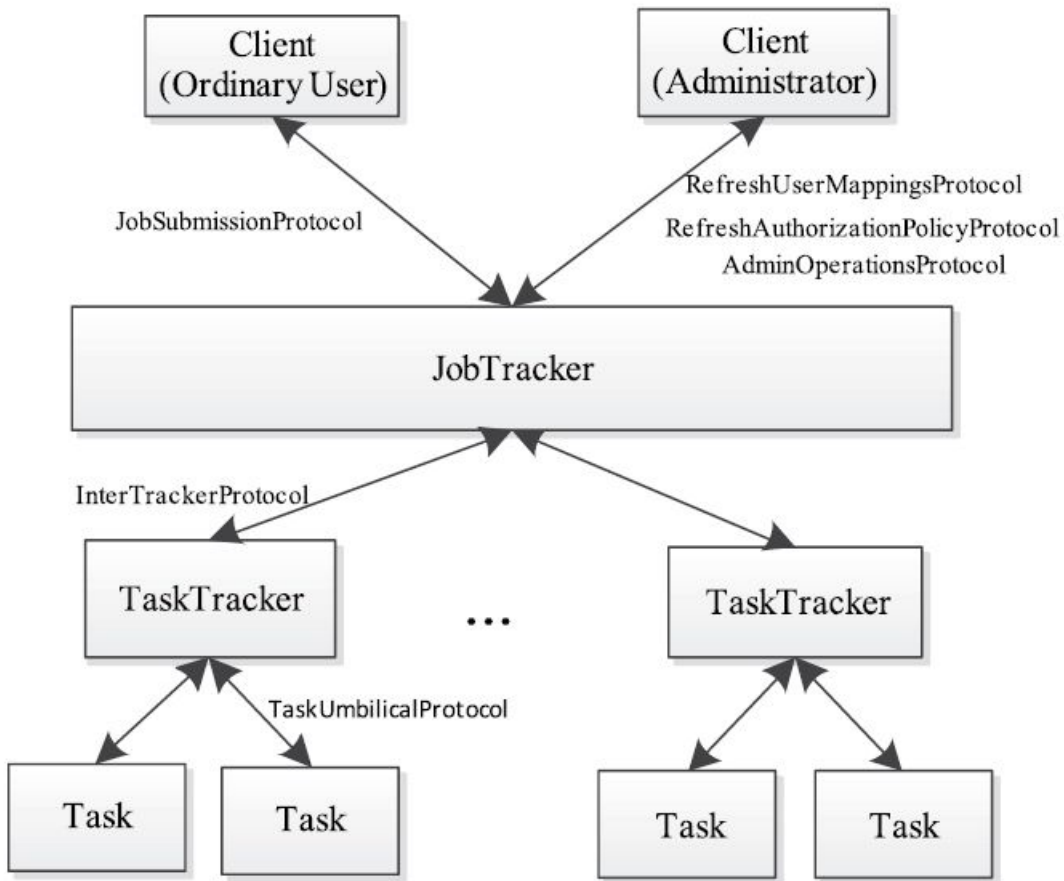


图 4-16 Hadoop MapReduce通信协议概览

另外，两个通信协议位于MapReduce框架内部，如下。

❑ **InterTrackerProtocol**: TaskTracker与JobTracker之间的通信协议。TaskTracker通过相关接口汇报本节点的资源使用情况和任务运行状态等信息，并执行JobTracker发送的命令。

❑ **TaskUmbilicalProtocol**: Task与TaskTracker之间的通信协议。每个Task实际上是其同节点TaskTracker的子进程，它们通过该协议汇报Task运行状态、运行进度等信息。

在Hadoop中，所有使用Hadoop RPC的协议基类均为VersionedProtocol。该类主要用于描述协议版本号，以防止不同版本号的客户端与服务端之间通信。在Hadoop MapReduce中，这六个通信协议与JobTracker, TaskTracker的类关系如图4-17所示。

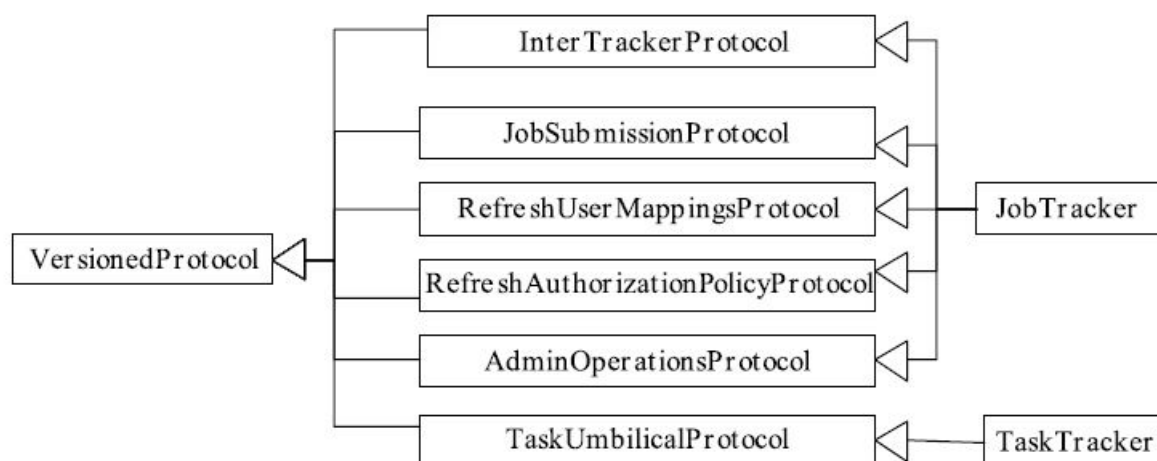


图 4-17 Hadoop MapReduce通信协议类关系图

[1] Hadoop 0.21.0以及之后的版本已将该协议名称改成ClientProtocol。

4.4.2 JobSubmissionProtocol通信协议

JobSubmissionProtocol是Client与JobTracker之间的通信协议。用户可通过该协议提交作业和查看作业运行状态。该协议中的接口可分为三类：

（1）作业提交

Client可通过以下RPC函数提交作业：

```
public JobStatus submitJob (JobID jobName, String jobSubmitDir, Credentials ts) throws IOException;
```

其中，jobName为该作业ID，Client可通过getNewJobId()函数为作业获取一个唯一的ID；jobSubmitDir为作业文件（如jar包，xml文件等）所在的目录，一般为HDFS上的一个目录；ts是该作业分配到的密钥或者安全令牌。

（2）作业控制

当用户提交作业之后，可进一步控制该作业，主要有三个操作：修改其作业优先级（setJobPriority函数）、杀死一个作业（killJob函数）、杀死一个任务（killTask函数）。

（3）查看系统状态和作业运行状态

该协议提供了一系列函数以供Client查看集群状态。下面是其中几个函数的声明：

```
//获取集群当前状态，如slot总数，所有正在运行的Task数目等
public ClusterStatus getClusterStatus (boolean detailed) throws IOException;
//获取某个作业的运行状态
public JobStatus getJobStatus (JobID jobid) throws IOException;
//获取所有作业的运行状态
public JobStatus[] getAllJobs () throws IOException;
```

4.4.3 InterTrackerProtocol通信协议

InterTrackerProtocol是TaskTracker与JobTracker之间的通信协议。TaskTracker通过该协议向JobTracker汇报所在节点的资源使用情况和任务运行状况，并接收和执行JobTracker返回的命令。

该协议中最重要的一个方法是heartbeat。它周期性地被调用，进而形成了TaskTracker与JobTracker之间的心跳。其定义如下：

```
HeartbeatResponse heartbeat (TaskTrackerStatus status,
boolean restarted,
boolean initialContact,
boolean acceptNewTasks,
short responseId)
throws IOException;
```

该函数的一个重要输入参数为TaskTrackerStatus类型的status。它封装了所在节点的资源使用情况（物理内存和虚拟内存总量和使用量，CPU个数以及利用率等）和任务运行情况（每个任务运行进度，状态以及所处的阶段等）。函数返回值类型为HeartbeatResponse。它包含了一个TaskTrackerAction类型的数组。该数组包含了JobTracker向TaskTracker传达的各种命令（我们将在7.4节详细介绍这几个命令），主要分为以下几种类型。

❑CommitTaskAction: Task运行完成，提交其产生的结果。

❑ReinitTrackerAction: 重新对自己（TaskTracker）初始化。

❑KillJobAction: 杀死某个作业，并清理其使用的资源。

❑KillTaskAction: 杀死某个任务。

❑LaunchTaskAction: 启动一个新任务。

该协议中其他几个均是getter方法，用于从JobTracker中获取信息：

```
//从JobTracker中获取某个作业已经完成的Task列表，这主要是为Reduce Task获取已完成的
Map Task列表，以便开始远程拷贝（shuffle）数据
TaskCompletionEvent[]getTaskCompletionEvents (JobID jobId, int fromEventId
, int maxEvents) throws IOException;
//获取JobTracker指定的系统目录，以便TaskTracker将作业相关的文件存放到该目录中
public String getSystemDir();
//获取JobTracker编译版本号，TaskTracker与JobTracker编译版本号一致才可启动
public String getBuildVersion()throws IOException;
```

4.4.4 TaskUmbilicalProtocol通信协议

TaskUmbilicalProtocol是Task与TaskTracker之间的通信协议。每个Task通过该协议向对应的TaskTracker汇报自己的运行状况或者出错信息。

按照调用频率，可将该协议中的方法分为两类：一类是周期性被调用的方法，另一类是按需调用的方法。其中，第一类方法主要有以下两个：

```
//Task向TaskTracker汇报自己的当前状态，状态信息被封装到TaskStatus中
boolean statusUpdate (TaskAttemptID taskId, TaskStatus taskStatus,
JvmContext jvmContext) throws IOException, InterruptedException;
//Task周期性探测TaskTracker是否活着
boolean ping (TaskAttemptID taskId, JvmContext jvmContext) throws IOException;
```

这两个方法并不是相互独立的，它们之间相互合作，共同完成Task状态汇报的任务。一般情况下，Task每隔3s会调用一次statusUpdate函数向TaskTracker汇报最新进度。然而，如果Task在3s内没有处理任何数据（比如当前记录处理速度太慢），则不再汇报进度，而是直接调用ping方法探测TaskTracker，以确保当前数据处理过程中它一直是活的。

第二类方法在Task的不同运行阶段被调用，其调用时机依次为：

（1）Task初始化

TaskTracker从JobTracker那里接收到一个启动新Task的命令（LaunchTaskAction）后，首先创建一个子进程（Child），并由该子进程调用getTask方法领取对应的Task。

（2）Task运行中

□汇报错误及异常：Task运行过程中可能会出现各种异常或者错误，而reportDiagnosticInfo/fsError/fatalError方法则分别用以汇报出现的Exception/FSError/Throwable异常和错误。对于Reduce Task而言，还提供了shuffleError方法汇报Shuffle阶段出现的错误。

□汇报记录范围：Hadoop可通过跳过坏记录提高程序的容错性（具体参考6.5.4节）。为了便于定位坏记录的位置，Task需要通过reportNextRecordRange方法不断向TaskTracker汇报将要处理的记录范围。

□获取Map Task完成列表：在MapReduce框架中，Reduce Task与Map Task之间存在数据依赖关系。该协议专门为Reduce Task提供了getMapCompletionEvent方法，以方便其从TaskTracker获取已经完成的Map Task列表，进而能够获取Map Task产生的临时数据存放位置，并远程读取（对应Reduce Task的Shuffle阶段）这些数据。

（3）Task运行完成

当Task处理完最后一条记录后，会依次调用commitPending、canCommit和done三个方法完成最后的收尾工作。

4.4.5 其他通信协议

其他三个通信协议，即`RefreshUserMappingsProtocol`、`RefreshAuthorizationPolicyProtocol`和`AdminOperationsProtocol`，均用于动态更新Hadoop MapReduce的相关配置文件。这些配置文件涉及对Hadoop某些模块的访问权限，因而往往只将其使用权限授予一些级别较高的用户（比如Hadoop管理员）。

（1）RefreshUserMappingsProtocol协议

该协议有两个作用：更新用户-用户组映射关系和更新超级用户代理列表，分别对应以下两个方法：

```
public void refreshUserToGroupsMappings() throws IOException;
public void refreshSuperUserGroupsConfiguration() throws IOException;
```

为了方便用户执行以上两个操作，Hadoop对其进行了封装，用户直接使用相应的Shell命令即可完成更新操作。

默认情况下，Hadoop使用UNIX/Linux系统中自带的用户-用户组映射关系，且对其进行了缓存。如果管理员在UNIX中修改了某些用户的映射关系，则可使用以下Shell命令更新到Hadoop MapReduce缓存中：

```
bin/hadoop mradmin-refreshUserToGroupsMappings
```

Hadoop提供了一种代理机制，允许某些用户伪装成其他用户执行某些操作。用户可在`core-site.xml`配置文件中添加以下配置选项：

```
<property>
<name>hadoop.proxyuser.oozie.groups</name>
<value>group1, group2</value>
<description>超级用户oozie可伪装成分组group1和group2中的任何用户</description>
</property>
<property>
<name>hadoop.proxyuser.oozie.hosts</name>
<value>host1, host2</value>
<description>超级用户oozie只能在host1和host2两个节点上进行伪装</description></property>
```

经过以上配置后，只要用户oozie拥有Hadoop Kerberos key（具体参考第11章），则host1和host2两个节点上group1和group2两个组中的所有用户可统一以用户oozie的身份提交作业。

管理员可动态修改这种超级用户代理列表，并通过以下命令完成更新操作：

```
bin/hadoop mradmin-refreshSuperUserGroupsConfiguration
```

（2）RefreshAuthorizationPolicyProtocol协议

该协议用于更新MapReduce服务级别访问控制列表，其中“服务级别访问控制列表”实际上是每个通信协议的访问控制列表。每种通信协议均对应一定的访问权限，比如，拥有JobSubmissionProtocol协议访问权限的用户可以提交作业，查看作业运行情况等。每个协议的访问控制列表可在配置文件`hadoop-policy.xml`中配置，比如：

```
<property>
<name>security.job.submission.protocol.acl</name>
<value>user1, user2 group1, group2</value>
</property>
```

经过以上配置后，用户user1、user2，用户组group1、group2拥有了JobSubmissionProtocol协议的访问权限，他们可以向Hadoop提交作业、查看作业运行情况等。

管理员可以直接通过以下命令动态更新配置文件`hadoop-policy.xml`：

```
bin/hadoop mradmin-refreshServiceAcl
```

该命令最终会调用RefreshAuthorizationPolicyProtocol协议中的refreshServiceAcl方法完成更新操作。

(3) AdminOperationsProtocol协议

该协议提供了用于更新队列访问控制列表（refreshQueues）和更新节点列表（refreshNodes）的两个方法。

□更新队列访问控制列表：Hadoop以队列为单位管理用户，管理员可配置队列与操作系统用户和用户组之间的映射关系，每个队列对应的用户或者用户组称为该队列的访问控制列表。在Hadoop中，队列访问控制列表信息可能存在于配置文件mapred-queue-acls.xml和调度器配置文件中。在配置文件mapred-queue-acls.xml中，管理员可为每个队列配置两种权限：提交作业权限和管理作业权限。对于某个用户而言，拥有某个队列的作业提交权限意味着该用户可向该队列提交作业并使用该队列中的计算资源，而拥有队列的作业管理权限意味着该用户可以改变任何作业的优先级、杀死任何作业等，比如：

```
<property>
<name>mapred.queue.myqueue.acl-submit-job</name>
<value>user1, user2 group1</value>
<description>用户user1、user2和用户组group1可向队列myqueue中提交作业</description>
</property>
<property>
<name>mapred.queue.myqueue.acl-administer-job</name>
<value>user1</value>
<description>用户user1可管理队列myqueue</description>
</property>
```

管理员可通过以下命令动态加载相关配置文件：

```
bin/hadoop mradmin-refreshQueues
```

□更新节点列表：Hadoop允许管理员为节点建立白名单（完全可以信赖的节点列表）和黑名单（不允许加入Hadoop集群的节点列表）。其中，白名单可通过配置选项mapred.hosts（在mapred-site.xml中）指定，而黑名单可通过mapred.hosts.exclude配置选项指定（在mapred-site.xml中）。管理员可通过以下命令动态更新这两个名单：

```
bin/hadoop mradmin-refreshNodes
```

4.5 小结

Hadoop RPC是Hadoop多个子系统公用的网络通信模块。其性能和可扩展性直接影响其上层系统的性能和可扩展性，因此扮演着极其重要的角色。

Hadoop RPC分为两层：上层是直接供外面使用的公共RPC接口；下层是一个客户机/服务器模型，该模型在实现过程中用到了Java自带的多个工具包，包括`java.lang.reflect`（反射机制和动态代理相关类）、`java.net`（网络编程库）和`java.nio`（NIO）等。

Hadoop RPC主要由三个大类组成，分别是RPC、Client和Server，分别对应对外编程接口、客户端实现和服务器端实现。其中，Server具有高性能和良好的可扩展性等特点，在具体实现时采用了线程池、事件驱动和Reactor设计模式等机制。

Hadoop MapReduce基于RPC框架实现了6个通信协议，分别是JobSubmissionsProtocol, RefreshUserMappingsProtocol, RefreshAuthorizationPolicyProtocol, AdminOperationsProtocol, InterTrackerProtocol和TaskUmbilicalProtocol。这些协议像是系统的“骨架”，支撑起整个MapReduce系统。

第5章 作业提交与初始化过程分析

前面一章介绍了Hadoop RPC框架，这是MapReduce运行时环境的基础。从本章开始，我们将以“作业的生命周期”为轴线介绍MapReduce内部实现原理。

在本章中，我们将深入分析一个MapReduce作业的提交与初始化过程，即从用户输入提交作业命令到作业初始化的整个过程。该过程涉及JobClient、JobTracker和TaskScheduler三个组件，它们的功能分别是准备运行环境、接收作业以及初始化作业。我们将在本章中深入分析这三个组件。

5.1 作业提交与初始化概述

总体而言，作业提交过程比较简单，它主要为后续作业执行准备环境，主要涉及创建目录、上传文件等操作；而一旦用户提交作业后，JobTracker端便会对接作业进行初始化。作业初始化的主要工作是根据输入数据量和作业配置参数将作业分解成若干个Map Task以及Reduce Task，并添加到相关数据结构中，以等待后续被调度执行。总之，可将作业提交与初始化过程分为四个步骤，如图5-1所示。

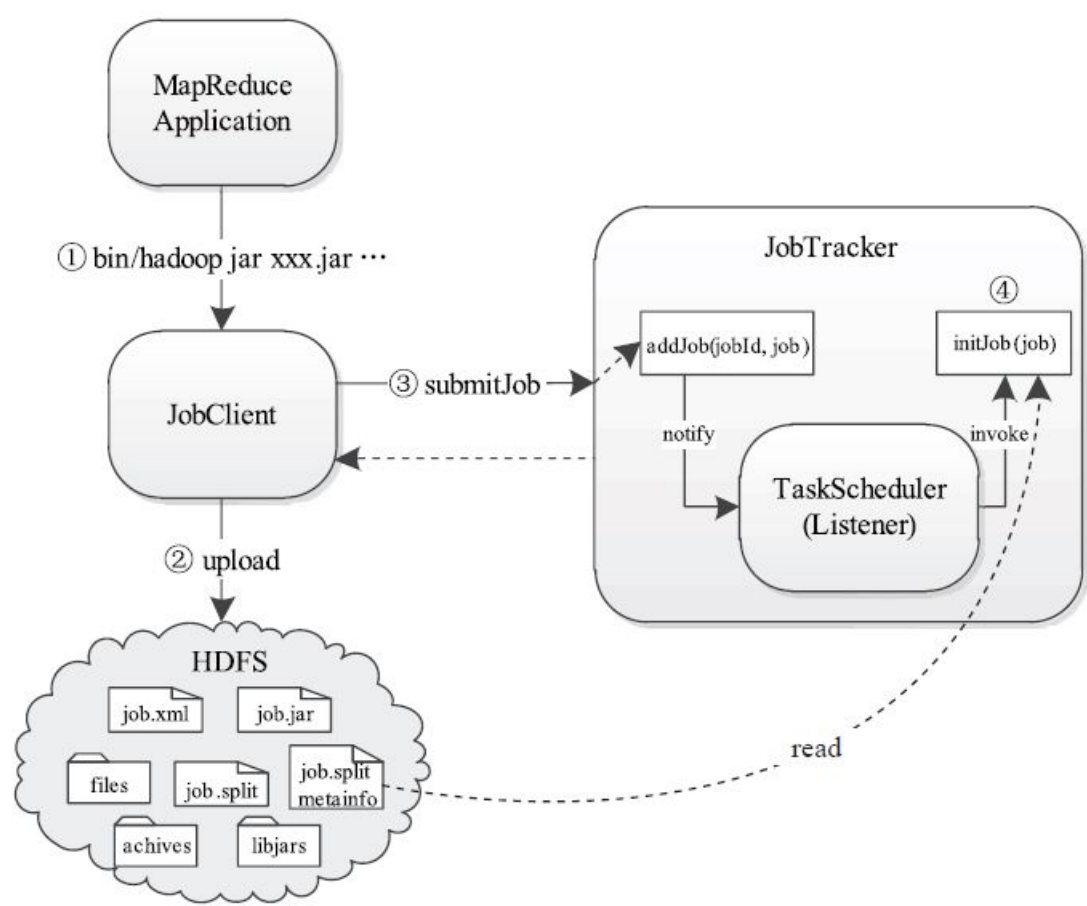


图 5-1 MapReduce作业提交与初始化过程

- 步骤1 用户使用Hadoop提供的Shell命令提交作业。
- 步骤2 JobClient按照作业配置信息（JonConf）将作业运行需要的全部文件上传到JobTracker文件系统（通常为HDFS，本书统一采用HDFS）的某个目录下。
- 步骤3 JobClient调用RPC接口向JobTracker提交作业。
- 步骤4 JobTracker接收到作业后，将其告知TaskScheduler，由TaskScheduler对作业进行初始化。

在步骤2中，之所以将作业相关文件（包括应用程序jar包、xml文件及其依赖的文件等，后面简称“作业文件”）上传到HDFS上，主要是出于以下两点考虑：

□HDFS是一个分布式文件系统，Hadoop集群中任何一个节点可以直接从该系统中下载文件。也就是说，HDFS上所有文件都是节点间共享的（不考虑文件权限）。

□作业文件是运行Task所必需的。它们一旦被上传到HDFS上后，任何一个Task只需知道存放路径便可以下载到自己的工作目录中使用，因而可看作一种非常简便的文件共享方式。

在接下来的几节中，我们将详细分析每个步骤中涉及的技术细节。

5.2 作业提交过程详解

在第3章中，我们介绍了Hadoop提供的三种应用程序开发方法，包括原始Java API、Hadoop Streaming和Hadoop Pipes，考虑到后两种方法的底层实际上均调用了Java API，因此，在本节中，我们以Java MapReduce程序为例讲解作业提交过程。

5.2.1 执行Shell命令

假设用户采用Java语言编写了一个MapReduce程序，并将其打包成xxx.jar，然后通过以下命令提交作业：

```
$HADOOP_HOME/bin/hadoop jar xxx.jar\  
-D mapred.job.name="xxx"\  
-D mapred.reduce.tasks=2\  
-files=blacklist.txt, whitelist.txt\  
-libjars=third-party.jar\  
-archives=dictionary.zip\  
-input/test/input\  
-output/test/output
```

当用户输入以上命令后，bin/hadoop脚本根据“jar”命令将作业交给RunJar类处理，相关shell代码如下：

```
.....  
elif["$COMMAND"="jar"]; then  
CLASS=org.apache.hadoop.util.RunJar  
.....
```

RunJar类中的main函数经解压jar包和设置环境变量后，将运行参数传递给MapReduce程序，并运行之。

用户的MapReduce程序已经配置了作业运行时需要的各种信息（如Mapper类，Reducer类，Reduce Task个数等），它最终在main函数中调用JobClient.runJob函数（新MapReduce API则使用job.waitForCompletion（true）函数）提交作业，这之后会依次经过图5-2所示的函数调用顺序才会将作业提交到JobTracker端。

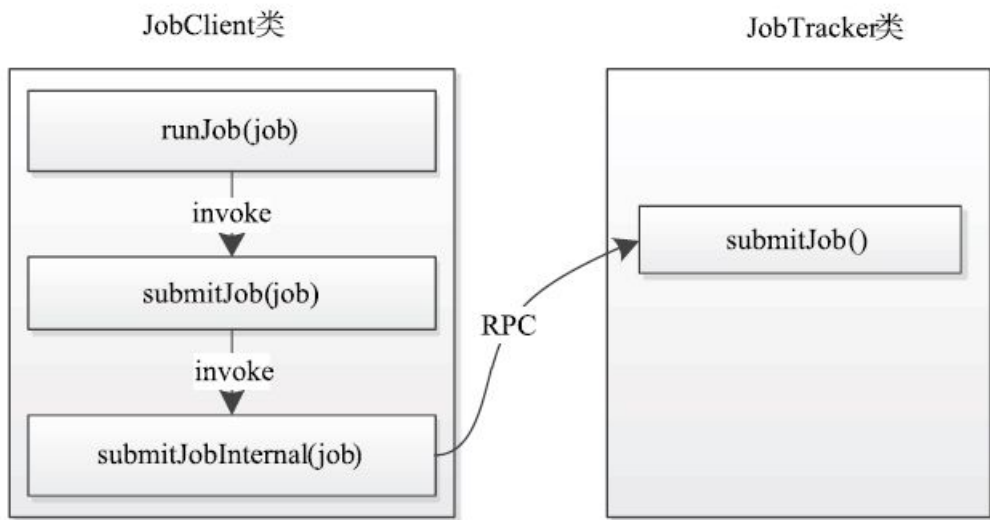


图 5-2 作业提交过程中函数调用关系

5.2.2 作业文件上传

JobClient将作业提交到JobTracker端之前，需要进行一些初始化工作，包括：获取作业ID，创建HDFS目录，上传作业文件以及生成Split文件等。这些工作由函数JobClient.submitJobInternal（job）实现，具体流程如图5-3所示。在本小节中，我们将重点分析文件上传过程，而生成InputSplit文件的相关细节将放在下一小节中讨论。

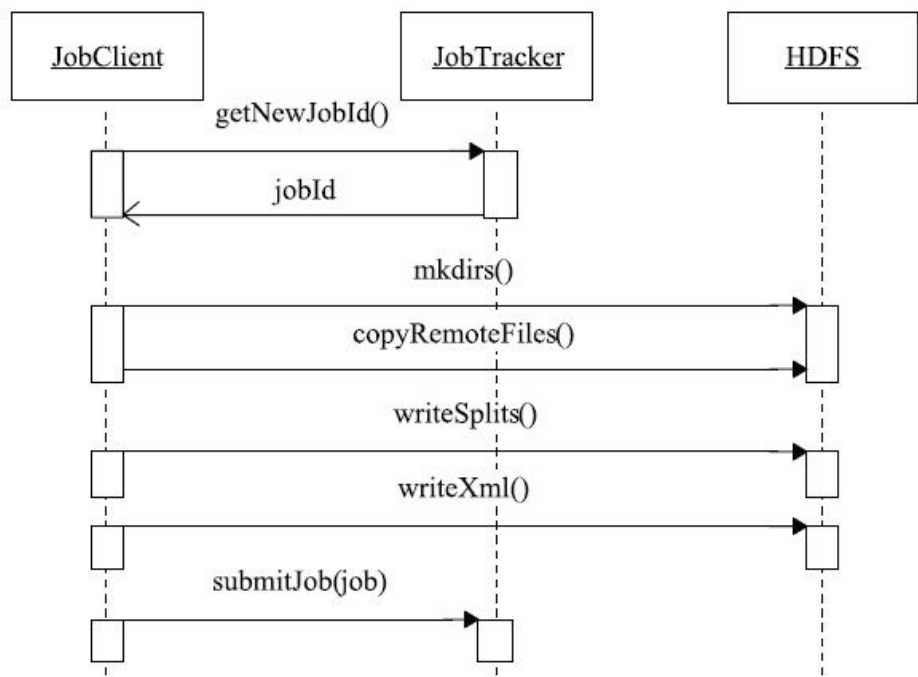


图 5-3 作业提交过程时序图

MapReduce作业文件的上传与下载是由DistributedCache工具完成的。它是Hadoop为方便用户进行应用程序开发而设计的数据分发工具。其整个工作流程对用户而言是透明的，也就是说，用户只需在提交作业时指定文件位置，至于这些文件的分发（需广播到各个TaskTracker上以运行Task），完全由DistributedCache工具完成，不需要用户参与。本小节主要涉及文件上传过程，而文件下载的相关细节将在5.4节中介绍。

通常而言，对于一个典型的Java MapReduce作业，可能包含以下资源。

- ❑程序jar包：用户用Java编写的MapReduce应用程序jar包。
- ❑作业配置文件：描述MapReduce应用程序的配置信息（根据JobConf对象生成的xml文件）。
- ❑依赖的第三方jar包：应用程序依赖的第三方jar包，提交作业时用参数“-libjars”指定。
- ❑依赖的归档文件：应用程序中用到多个文件，可直接打包成归档文件（通常为一些压缩文件），提交作业时用参数“-archives”指定。
- ❑依赖的普通文件：应用程序中可能用到普通文件，比如文本格式的字典文件，提交作业时用参数“-files”指定。

注意 应用程序依赖的文件可以存放在本地磁盘上，也可以存放在HDFS上，默认情况下是存放在本地磁盘上的，如上一小节中作业提交命令参数“-libjars=third-party.jar”指定的third-party.jar文件便存在于本地目录。如果程序依赖的文件已经事先上传到HDFS上，比如目录/data/中，则可以使用参数“-libjars=hdfs: ///data/third-party.jar”指定。

上述所有文件在JobClient端被提交到HDFS上，涉及的父目录如表5-1所示。

表 5-1 作业文件在 HDFS 中涉及的父目录

作业属性	属性值	说 明
mapreduce.jobtracker.staging.root.dir	<code>\${hadoop.tmp.dir}/mapred/staging</code>	HDFS 上作业文件的上传目录，由管理员配置
mapreduce.job.dir	<code>\${mapreduce.jobtracker.staging.root.dir}/\${user}/.staging/\${jobId}</code>	用户 <code>\${user}</code> 的作业 <code>{jobId}</code> 相关文件存放目录

5.2.1节中MapReduce作业在HDFS中的目录组织结构如图5-4所示。

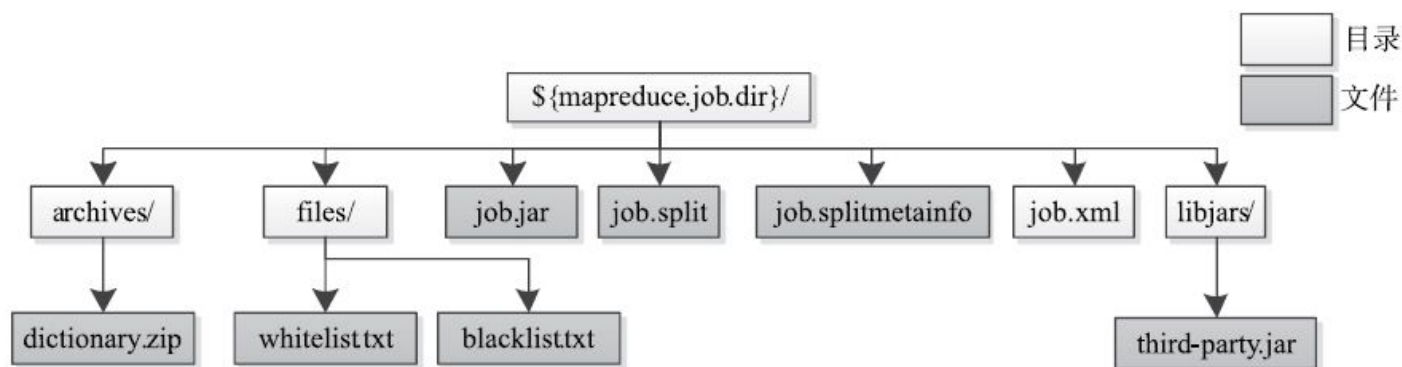


图 5-4 一个应用程序在HDFS上的目录组织结构

在这个例子中，由于参数`-libjars`、`-archives`和`-files`指定的文件均属于本地文件，因而`JobClient`会将这些文件上传到HDFS上；如果有些文件已经存在于HDFS上，则不再需要上传。

文件上传完毕后，会将这些目录信息保存到作业配置对象`JobConf`中，其对应的作业属性如表5-2所示。

表 5-2 作业文件上传后添加的新配置选项

作业属性	说 明
mapred.cache.files	作业依赖的普通文件在 HDFS 上的存放路径
mapred.job.classpath.archives	作业依赖的 jar 包在 HDFS 上的存放路径
mapred.cache.archives	作业依赖的压缩文件在 HDFS 上的存放路径
mapreduce.job.cache.files.visibilities	作业依赖的普通文件的可见性。如果是 public 可见性，则为 true，否则为 false
mapreduce.job.cache.archives.visibilities	作业依赖的归档文件的可见性。如果是 public 级别的可见性，则为 true，否则为 false
mapred.cache.files.timestamps	作业依赖的普通文件的最后一次修改时间的时间戳
mapred.cache.archives.timestamps	作业依赖的压缩文件的最后一次修改时间的时间戳
mapred.cache.files.filesizes	作业依赖的普通文件的大小
mapred.cache.archives.filesizes	作业依赖的归档文件的大小
mapred.jar	用户应用程序 jar 路径

`DistributedCache`将文件分为两种可见级别，分别是`private`级别和`public`级别。其中，`private`级别文件只会被当前用户使用，不能与其他用户共享；而`public`级别文件则不同，它们在每个节点上保存一份，可被本节点上的所有作业和用户共享，这样可以大大降低文件复制代价，提高了作业运行效率。一个文件/目录要成为`public`级别文件/目录，需同时满足以下两个条件：

- 该文件/目录对所有用户/用户组均有可读权限。
- 该文件/目录的父目录、父目录的父目录……对所有用户/用户组有可执行权限。

作业文件上传到HDFS后，可能会有大量节点同时从HDFS上下载这些文件，进而产生文件访问热点现象，造成性能瓶颈。为此，JobClient上传这些文件时会调高它们的副本数（由参数`mapred.submit.replication`指定，默认是10）以通过分摊负载方式避免产生访问热点。

5.2.3 产生InputSplit文件

用户提交MapReduce作业后，JobClient会调用InputFormat的getSplits方法生成InputSplit相关信息。该信息包括两部分：InputSplit元数据信息和原始InputSplit信息。其中，第一部分将被JobTracker使用，用以生成Task本地性（Task Locality）相关的数据结构；而第二部分则将被Map Task初始化时使用，用以获取自己要处理的数据。这两部分信息分别被保存到目录\${mapreduce.jobtracker.staging.root.dir}/\${user}/.staging/\${JobId}下的文件job.split和job.splitmetainfo中。

InputSplit相关操作放在包org.apache.hadoop.mapreduce.split中，主要包含三个类JobSplit、JobSplitWriter和SplitMetaInfoReader。它们的关系如图5-5所示。

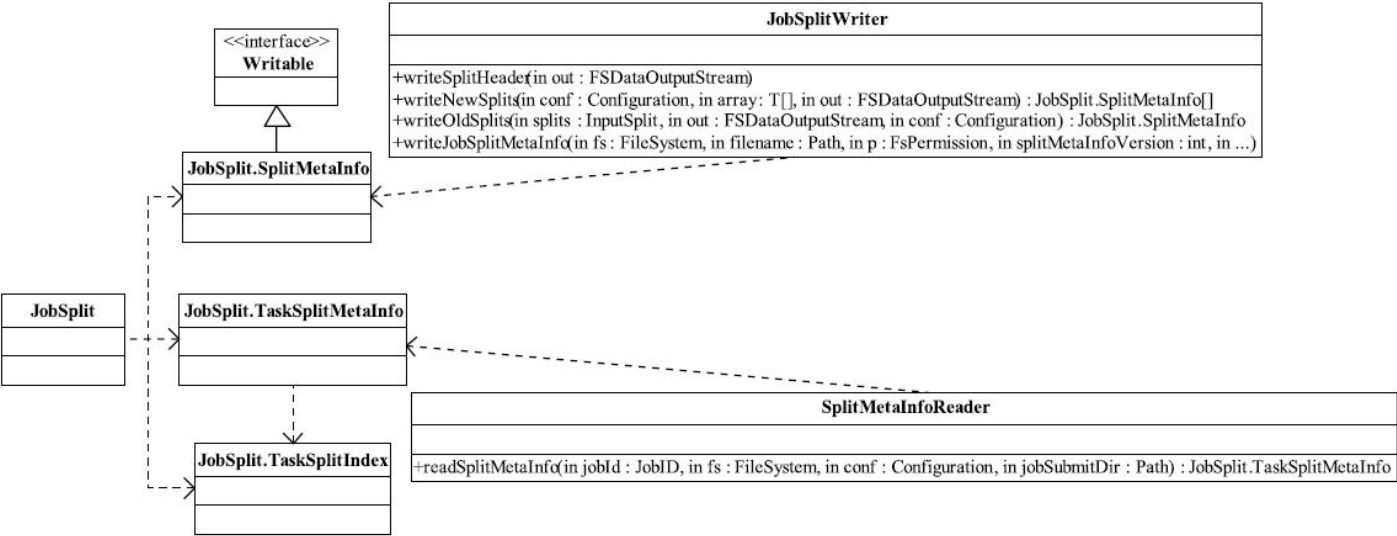


图 5-5 InputSplit类关系图

JobSplit封装了读写InputSplit相关的基础类，主要包括以下三个。

□ SplitMetaInfo: 描述一个InputSplit的元数据信息，包括以下三项内容：

```
private long startOffset; //该InputSplit元信息在job.split文件中的偏移量
private long inputDataLength; //该InputSplit的数据长度
private String[] locations; //该InputSplit所在的host列表
```

所有InputSplit对应的SplitMetaInfo将被保存到文件job.splitmetainfo中。该文件内容组织方式如图5-6所示，内容依次为：一个用于标识InputSplit元数据文件头部的字符串“META-SP”，文件版本号splitVersion（当前值是1），作业对应的InputSplit数目length，最后是length个InputSplit对应的SplitMetaInfo信息。

META-SP	splitVersion	length
SplitMetaInfo		
SplitMetaInfo		
...		

图 5-6 job.splitmetainfo文件内容组织方式

作业在JobTracker端初始化时，需读取job.splitmetainfo文件创建Map Task，具体参考5.3节。

□ TaskSplitMetaInfo: 用于保存InputSplit元信息的数据结构，包括以下三项内容：

```
private TaskSplitIndex splitIndex; //Split元信息在job.split文件中的位置
private long inputDataLength; //InputSplit的数据长度
private String[] locations; //InputSplit所在的host列表
```

这些信息是在作业初始化时，JobTracker从文件job.splitmetainfo中获取的。其中，host列表信息是任务调度器判断任务是否具有本地性的最重要因素，而splitIndex信息保存了新任务需处理的数据位置信息在文件job.split中的索引，TaskTracker（从JobTracker端）收到该信息后，便可以从job.split文件中读取InputSplit信息，进而运行一个新任务。

□ TaskSplitIndex: JobTracker向TaskTracker分配新任务时，TaskSplitIndex用于指定新任务待处理数据位置信息在文件job.split中的索引，主要包括两项内容：

```
private String splitLocation; //job.split文件的位置（目录）
private long startOffset; //InputSplit信息在job.split文件中的位置
```

5.2.4 作业提交到JobTracker

JobClient最终调用RPC方法submitJob将作业提交到JobTracker端，在JobTracker.submitJob中，会依次进行以下操作：

（1）为作业创建JobInProgress对象

JobTracker会为每个作业创建一个JobInProgress对象。该对象维护了作业的运行信息。它在作业运行过程中一直存在，主要用于跟踪正在运行作业的运行状态和进度。

（2）检查用户是否具有指定队列的作业提交权限

Hadoop以队列为单位管理作业和资源，每个队列分配有一定量的资源，每个用户属于一个或者多个队列且只能使用所属队列中的资源。第4章中提到，管理员可为每个队列指定哪些用户具有作业提交权限和管理权限。

（3）检查作业配置的内存使用量是否合理

用户提交作业时，可分别用参数mapred.job.map.memory.mb和mapred.job.reduce.memory.mb指定Map Task和Reduce Task占用的内存量；而管理员可通过参数mapred.cluster.max.map.memory.mb和mapred.cluster.max.reduce.memory.mb限制用户配置的任务最大内存使用量，一旦用户配置的内存使用量超过系统限制，则作业提交失败。

（4）通知TaskScheduler初始化作业

JobTracker收到作业后，并不会马上对其初始化，而是交给调度器，由它按照一定的策略对作业进行初始化。之所以不选择JobTracker而让调度器初始化，主要考虑到以下两个原因：

□作业一旦初始化后便会占用一定量的内存资源，为了防止大量初始化的作业排队等待调度而占用大量不必要的内存资源，Hadoop按照一定的策略选择性地初始化作业以节省内存资源；

□任务调度器的职责是根据每个节点的资源使用情况对其分配最合适的任务，而只有经过初始化的作业才有可能得到调度，因而将作业初始化策略嵌到调度器中是一种比较合理的设计。

Hadoop的调度器是一个可插拔模块，用户可通过实现TaskScheduler接口设计自己的调度器。当前Hadoop默认的调度器是JobQueueTaskScheduler。它采用的调度策略是先来先服务（First In First Out, FIFO）。另外两个比较常用的调度器是Fair Scheduler和Capacity Scheduler，具体参考第10章。

JobTracker采用了观察者设计模式（也称为发布-订阅模式）将“提交新作业”这一事件告诉TaskScheduler，如图5-7所示。

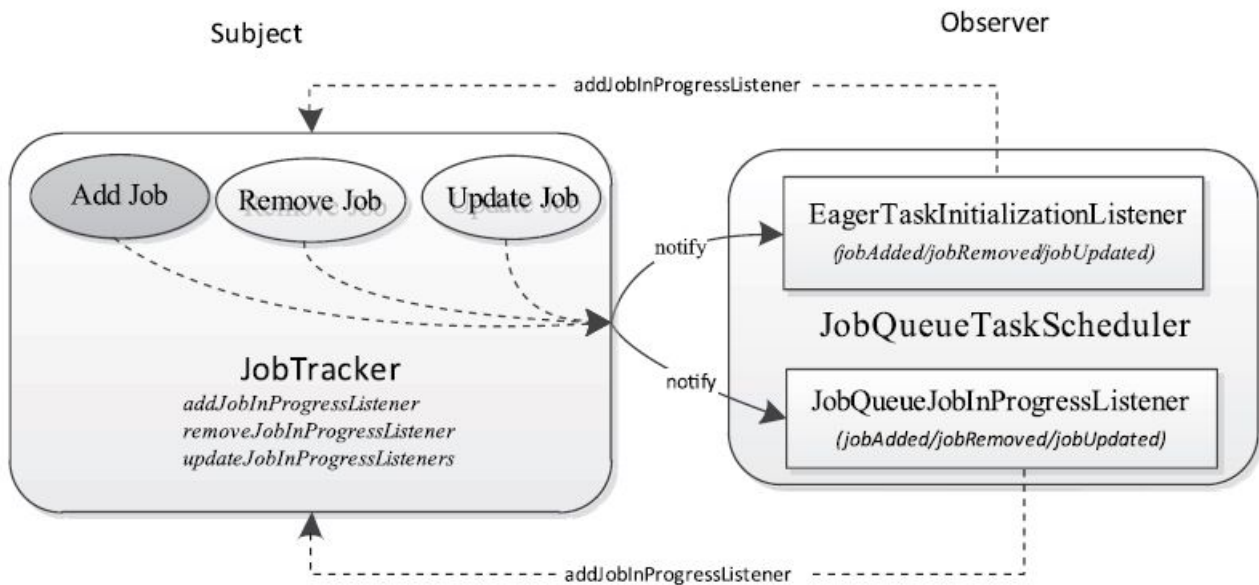


图 5-7 JobTracker采用观察者设计模式将作业变化（添加/删除/更新作业）通知TaskScheduler

JobTracker采用了观察者设计模式将“提交新作业”这一事件告诉TaskScheduler的相关代码如下：

```

private synchronized JobStatus addJob (JobID jobId, JobInProgress job)
throws IOException{
    .....
    synchronized (jobs) {
        synchronized (taskScheduler) {
            jobs.put (job.getProfile().getJobID(), job);
            for (JobInProgressListener listener: jobInProgressListeners) {
                listener.jobAdded (job); //依次通知每个已注册的JobInProgressListener
            }
        }
    }
    .....
}
  
```

JobTracker启动时会根据配置参数mapred.jobtracker.taskScheduler构造相应的任务调度器，并调用它的start()方法进行初始化。在该方法中，调度器会向JobTracker注册JobInProgressListener对象以监听作业的添加/删除/更新等事件。以默认调度器JobQueueTaskScheduler为例，它的start()方法如下：

```

public synchronized void start()throws IOException{
    super.start();
    //此处的taskTrackerManager实际上就是JobTracker对象，向JobTracker注册一个
    //JobQueueJobInProgressListener
    taskTrackerManager.addJobInProgressListener (jobQueueJobInProgressListener);
    eagerTaskInitializationListener.setTaskTrackerManager (taskTrackerManager);
    eagerTaskInitializationListener.start();
    //向JobTracker注册EagerTaskInitializationListener
    taskTrackerManager.addJobInProgressListener (
        eagerTaskInitializationListener);
}
  
```

在上面的代码中，JobQueueTaskScheduler向JobTracker注册了两个JobInProgress Listener: EagerTaskInitializationListener和JobQueueJobInProgressListener，它们分别用于作业初始化和作业排序（具体参考第10章）。需要注意的是，代码中的taskTrackerManager实际上就是JobTracker，其在JobTracker类中的相关代码如下：

```

public static JobTracker startTracker (JobConf conf, String identifier)
throws IOException, InterruptedException{
    .....
    result=new JobTracker (conf, identifier); //创建唯一的JobTracker实例
    result.taskScheduler.setTaskTrackerManager (result); //将JobTracker实例传递给TaskScheduler
    .....
}
  
```

5.3 作业初始化过程详解

调度器调用`JobTracker.initJob()`函数对新作业进行初始化。作业初始化的主要工作是构造Map Task和Reduce Task并对它们进行初始化。

如图5-8所示，Hadoop将每个作业分解成4种类型的任务，分别是Setup Task、Map Task、Reduce Task和Cleanup Task。它们的运行时信息由`TaskInProgress`类维护，因此，创建这些任务实际上是创建`TaskInProgress`对象。

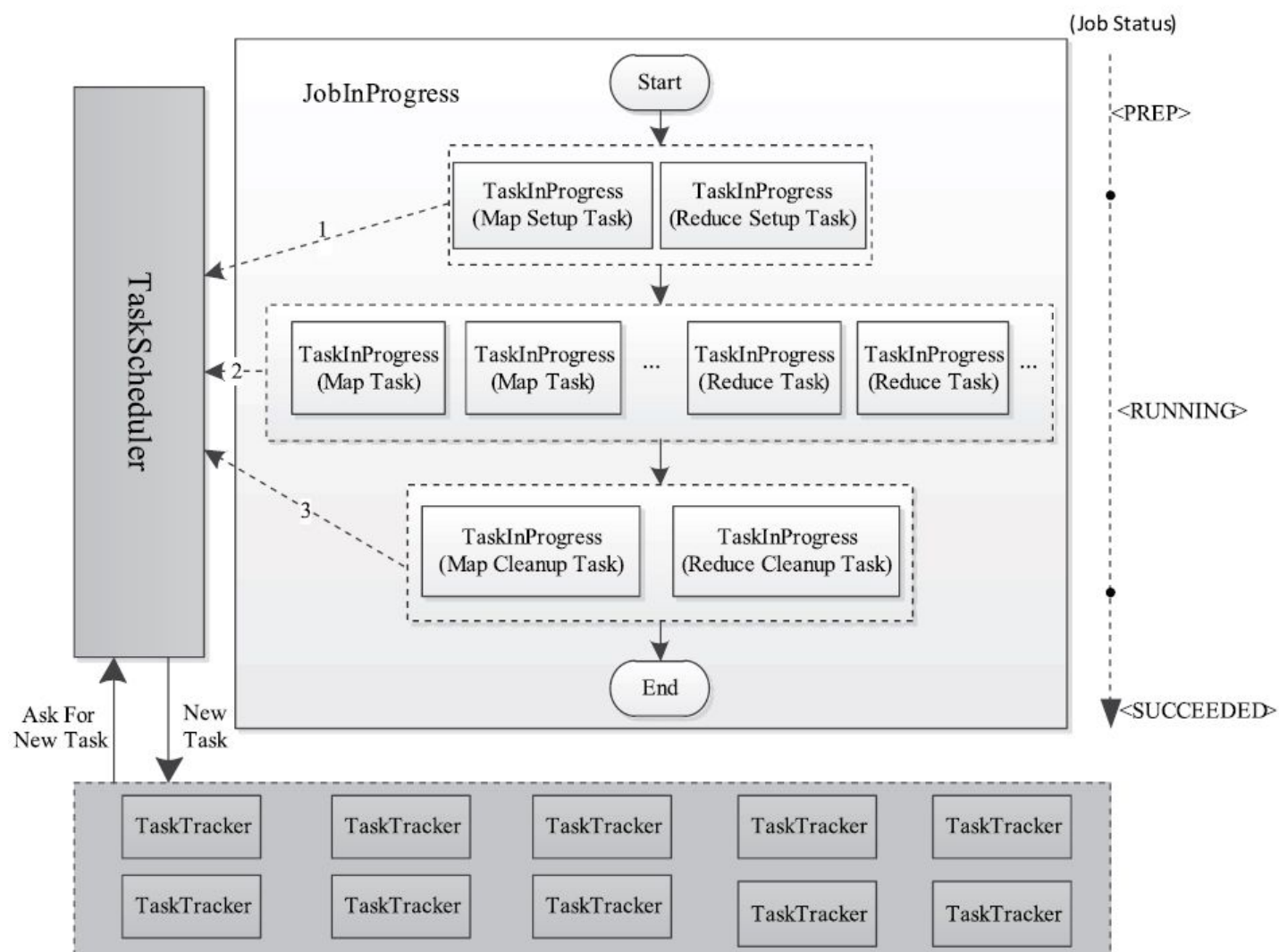


图 5-8 作业初始化过程概述

上述4种任务的作用及创建过程如下。

□ **Setup Task:** 作业初始化标识性任务。它进行一些非常简单的作业初始化工作，比如将运行状态设置为“setup”，调用 `OutputCommitter.setupJob()` 函数等。该任务运行完后，作业由PREP状态变为RUNNING状态，并开始运行Map Task。该类型任务又被分为Map Setup Task和Reduce Setup Task两种，且每个作业各有一个。它们运行时分别占用一个Map slot和Reduce slot。由于这两种任务功能相同，因此有且只有一个可以获得运行的机会（即只要有一个开始运行，另一个马上被杀掉，而具体哪一个能够运行，取决于当时存在的空闲slot种类及调度策略）。

创建该类任务的相关代码如下：

```
.....  
//创建两个TIP, Map和Reduce各一个  
setup=new TaskInProgress[2];  
//setup map tip.这个Map不会使用任何split, 只赋予它一个空split  
setup[0]=new TaskInProgress(jobId, jobFile, emptySplit,  
jobtracker, conf, this, numMapTasks+1, 1);
```

```
setup[0].setJobSetupTask();
//setup reduce tip.
setup[1]=new TaskInProgress(jobId, jobFile, numMapTasks,
numReduceTasks+1, jobtracker, conf, this, 1);
setup[1].setJobSetupTask();
```

□ **Map Task**: Map阶段处理数据的任务。其数目及对应的处理数据片段由应用程序中的InputFormat组件确定。

创建该类任务的相关代码如下:

```
//读取job.splitmetainfo文件, 还原TaskSplitMetaInfo对象, 每个对象描述了一个InputSplit信息
TaskSplitMetaInfo[] splits=createSplits(jobId);
numMapTasks=splits.length;
.....
maps=new TaskInProgress[numMapTasks];
for (int i=0; i<numMapTasks; ++i) {
inputLength+=splits[i].getInputDataLength();
maps[i]=new TaskInProgress(jobId, jobFile,
splits[i],
jobtracker, conf, this, i, numSlotsPerMap);
}
```

□ **Reduce Task**: Reduce阶段处理数据的任务。其数目由用户通过参数mapred.reduce.tasks (默认数目为1) 指定。考虑到Reduce Task能否运行依赖于Map Task的输出结果, 因此, Hadoop刚开始只会调度Map Task, 直到Map Task完成数目达到一定比例 (由参数mapred.reduce.slowstart.completed.maps指定, 默认是0.05, 即5%) 后, 才开始调度Reduce Task。

创建该类任务的相关代码如下:

```
this.reduce=new TaskInProgress[numReduceTasks];
for (int i=0; i<numReduceTasks; i++) {
reduces[i]=new TaskInProgress(jobId, jobFile,
numMapTasks, i,
jobtracker, conf, this, numSlotsPerReduce);
nonRunningReduces.add(reduces[i]);
}
```

□ **Cleanup Task**: 作业结束标志性任务, 主要完成一些作业清理工作, 比如删除作业运行过程中用到的一些临时目录 (比如临时目录)。一旦该任务运行成功后, 作业由RUNNING状态变为SUCCEEDED状态。

随着Hadoop的普及和衍化, 有人发现引入Setup/Cleanup Task会拖慢作业执行进度且降低作业的可靠性^[1], 这主要是因为Hadoop除了需保证每个Map/Reduce Task运行成功外, 还要保证Setup/Cleanup Task成功。对于Map/Reduce Task而言, 可通过推测执行机制 (具体参考6.6节) 避免出现“拖后腿”任务。然而, 由于Setup/Cleanup Task不会处理任何数据, 两种任务进度只有0%和100%两个值, 从而使得推测式任务机制对之不适用。为解决该问题, 从0.21.0版本开始, Hadoop将是否启用Setup/Cleanup Task变成了可配置的选项^[2], 用户可通过参数mapred.committer.job.setup.cleanup.needed配置是否为作业创建Setup/Cleanup Task。

这4种任务的调度顺序是Setup Task、Map Task、Reduce Task和Cleanup Task, 其中, Map Task完成一定比例后便开始调用Reduce Task。这期间涉及一些任务调度策略, 我们将在第6章中详细讨论。

[1] <https://issues.apache.org/jira/browse/MAPREDUCE-1099>

[2] <https://issues.apache.org/jira/browse/MAPREDUCE-463>

5.4 Hadoop DistributedCache原理分析

DistributedCache是Hadoop为方便用户进行应用程序开发而设计的文件分发工具。它能够将只读的外部文件自动分发到各个节点上进行本地缓存，以便Task运行时加载使用。它的大体工作流程如下：用户提交作业后，Hadoop将由-files和-archives选项指定的文件复制到JobTracker的文件系统（一般为HDFS）中；之后，当某个TaskTracker收到该作业的第一个Task后，该任务将负责从JobTracker文件系统中将文件下载到本地磁盘进行缓存，这样后续的Task就可以直接在本地访问这些文件了。除了文件分发外，DistributedCache还可用于软件自动安装部署。比如，用户使用PHP语言编写了MapReduce程序，为了能够让程序成功运行，用户要求运维人员在Hadoop集群的各个节点上提前安装好PHP解释器，而当需要升级PHP解释器时，可能需通知Hadoop运维人员进行一次升级，这使得软件升级变得非常麻烦。为了让软件升级变得更可控，用户可采用DistributedCache将PHP解释器分发到各个节点上，每次运行程序时，DistributedCache会检查PHP解释器被改过（比如升级新版本），如果是，则会自动重新下载。

本节首先介绍Hadoop DistributedCache使用场景和使用方法，接着介绍其工作原理。

5.4.1 使用方法介绍

用户编写的MapReduce应用程序往往需要一些外部的资源，比如分词程序需词表文件，或者依赖于三方的jar包。这时候，我们希望每个Task初始化时能够加载这些文件，而DistributedCache正是为了完成该功能而提供的。

使用Hadoop DistributedCache通常有两种方法：调用相关API和设置命令行参数。

（1）调用相关API

Hadoop DistributedCache允许用户分发归档文件（后缀为.zip、.jar、.tar、.tgz或者.tar.gz的文件）和普通文件，对应的API如下：

```
//添加归档文件
void addCacheArchive (URI uri, Configuration conf)
void setCacheArchives (URI[]archives, Configuration conf)
//添加普通文件
void addCacheFile (URI uri, Configuration conf)
void setCacheFiles (URI[]files, Configuration conf)
//将三方jar包或者动态库添加到classpath中
void addFileToClassPath (Path file, Configuration conf)
//在任务工作目录下建立文件软连接
void createSymlink (Configuration conf)
```

使用Hadoop DistributedCache可分为3个步骤：

步骤1 在HDFS上准备好文件（文本文件、压缩文件、jar包等），并按照文件可见级别设置目录/文件的访问权限；

步骤2 调用相关API添加文件信息，这里主要是配置作业的JobConf对象；

步骤3 在Mapper或者Reducer类中使用文件，Mapper或者Reducer开始运行前，各种文件已经下载到本地的工作目录中，直接调用文件读写API即可获取文件内容。

【实例】假设一个MapReduce应用程序需要dictionary.zip、blacklist.txt、whitelist.txt和third-party.jar四个文件，其中，dictionary.zip和third-party.jar为private可见级别，而blacklist.txt和whitelist.txt为public可见级别，则可按以下步骤分发这些文件：

步骤1 准备文件。将文件dictionary.zip和third-party.jar上传到HDFS上的目录/data/private/中，blacklist.txt和whitelist.txt上传到目录/data/public/中。其中，目录/data/private/的权限为“drwxr-xr--”，目录/data/public/的权限为“drwxr-xr-x”。

```
$bin/hadoop fs-copyFromLocal dictionary.zip/data/private/
$bin/hadoop fs-copyFromLocal third-party.jar/data/private/
$bin/hadoop fs-copyFromLocal blacklist.txt/data/public/
$bin/hadoop fs-copyFromLocal whitelist.txt/data/public/
```

步骤2 配置JobConf。

```
JobConf job=new JobConf();
DistributedCache.addCacheFile(new URI("/data/public/blacklist.txt#blacklist"), job);
DistributedCache.addCacheFile(new URI("/data/public/whitelist.txt#whitelist"), job);
DistributedCache.addFileToClassPath(new Path("/data/private/third-party.jar"), job);
DistributedCache.addCacheArchive(new URI("/data/private/dictionary.zip"), job);
DistributedCache.createSymlink(job);
```

步骤3 在Mapper或者Reducer类中使用文件。

```
public static class MapClass extends MapReduceBase
implements Mapper<K, V, K, V>{
private Path[]localArchives;
private Path[]localFiles;
public void configure (JobConf job) {
//在本地获取archives或者files
localArchives=DistributedCache.getLocalCacheArchives (job);
localFiles=DistributedCache.getLocalCacheFiles (job);
//调用文件API读取文件内容，保存到相关变量中
.....
}
public void map (K key, V value,
OutputCollector<K, V>output, Reporter reporter)
throws IOException{
//在此使用缓存中的archives/files
//.....
output.collect (k, v);
}
}
```

(2) 设置命令行参数

这是一种比较简单且灵活的方法，但前提是用户编写MapReduce应用程序时实现了Tool接口支持常规选项。该方法包括两个步骤，其中第一个步骤与“调用相关API”的步骤1相同，而第二个步骤则是使用以下两种Shell命令之一提交作业。

Shell命令1:

```
$HADOOP_HOME/bin/hadoop jar xxx.jar\
-files=hdfs:///data/public/blacklist.txt#blacklist, \
hdfs:///data/public/whitelist.txt#whitelist\
-libjars=hdfs:///data/private/third-party.jar\
-archives=hdfs:///data/private/dictionary.zip\
.....
```

Shell命令2:

```
$HADOOP_HOME/bin/hadoop jar xxx.jar\
-D mapred.cache.files=/data/public/blacklist.txt#blacklist, \
/data/public/whitelist.txt#whitelist\
-D mapred.cache.archives=/data/private/dictionary.zip\
-D mapred.job.classpath.files=/data/private/third-party.jar\
-D mapred.create.symlink=yes\
.....
```

5.4.2 工作原理分析

Hadoop DistributedCache工作原理如图5-9所示。它主要的功能是将作业文件分发到各个TaskTracker上，具体流程可分为4个步骤：

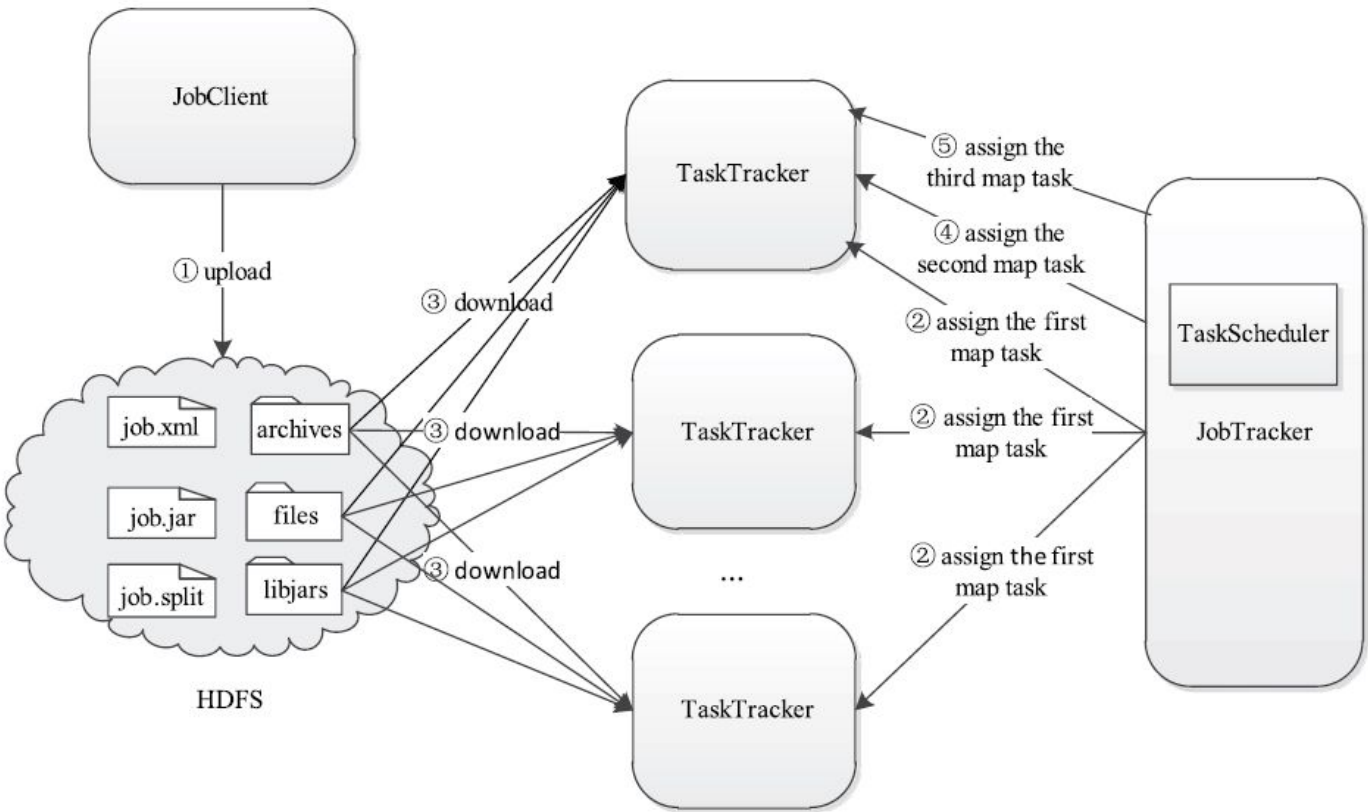


图 5-9 Hadoop DistributedCache工作原理图

步骤1 用户提交作业后，DistributedCache将作业文件上传到HDFS上的固定目录中，具体见5.2.2节。

步骤2 JobTracker端的任务调度器将作业对应的任务发派到各个TaskTracker上。

步骤3 任何一个TaskTracker收到该作业的第一个任务后，由DistributedCache自动将作业文件缓存到本地目录下（对于后缀为.zip、.jar、.tar、.tgz或者.tar.gz的文件，会自动对其进行解压缩），然后开始启动该任务。

步骤4 对于TaskTracker接下来收到的任务，DistributedCache不会再重复为其下载文件，而是直接运行。

下面分析TaskTracker中作业目录组织结构，具体如图5-10所示。在TaskTracker本地目录中，不同可见级别的文件被存放于不同的目录。对于public级别的文件，会被保存到公共目录`${mapred.local.dir}/taskTracker/distcache`中，该目录中的文件可被该TaskTracker上所有用户共享，也就是说，这些文件只会被下载一遍，后面的任何用户的作业可直接使用；对于private级别的文件，则被保存到用户私有目录`${mapred.local.dir}/taskTracker/${user}`下，在该目录下，将DistributedCache文件和作业运行需要文件分别放到子目录distcache和jobcache中，其中jobcache目录相当于作业的工作目录，它里面的文件大多是指向其他文件和目录的软连接，这些目录中的文件只能被该用户的作业共享。

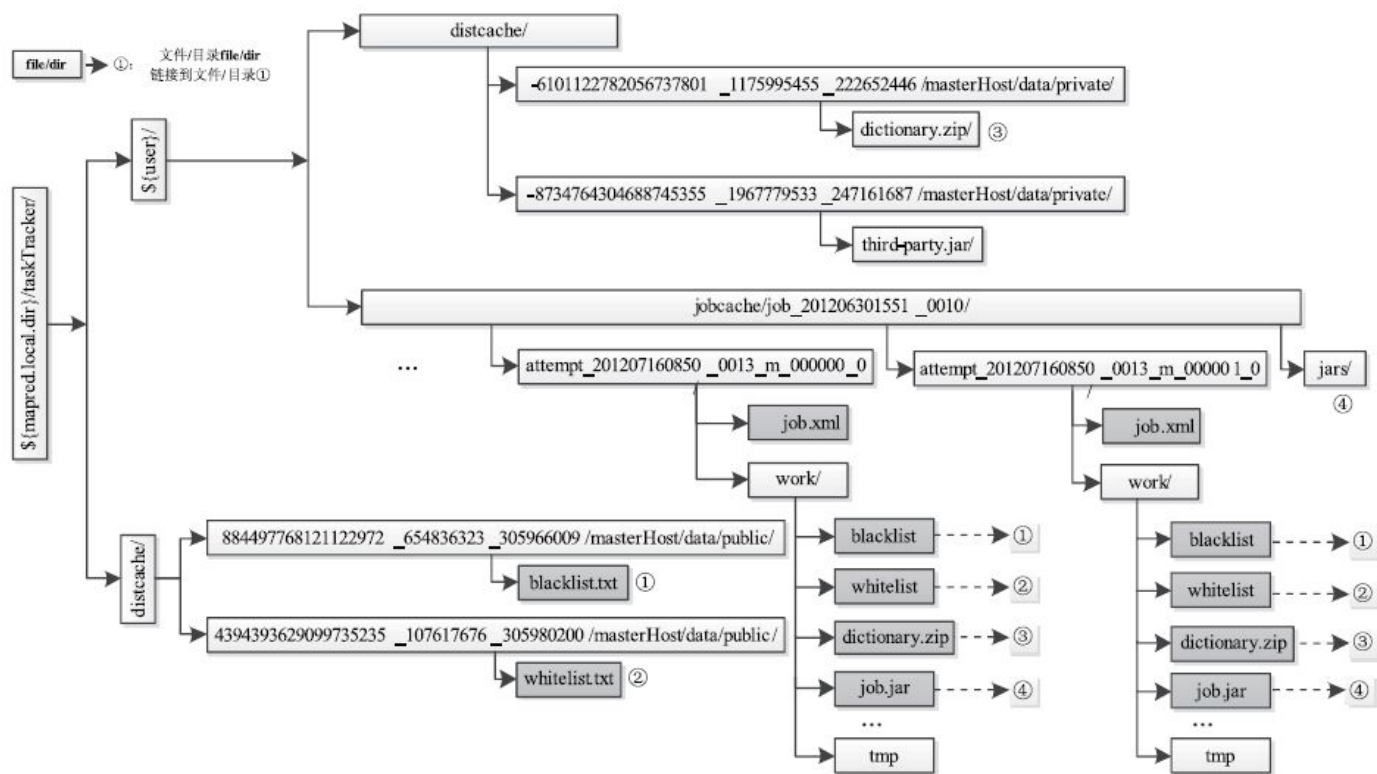


图 5-10 TaskTracker端作业目录组织结构

DistributedCache中的文件或者目录并不是用完后立即被清理的，而是由专门的一个线程根据文件大小上限（由参数`local.cache.size`设定，默认是10 GB）和文件/目录数目上限（由参数`mapreduce.tasktracker.local.cache.numberdirectories`设定，默认是10 000）周期性（由参数`mapreduce.tasktracker.distributedcache.checkperiod`设定，默认是60）地进行清理。

Hadoop DistributedCache的实现在包`org.apache.hadoop.filecache`中，主要包括DistributedCache、TaskDistributedCacheManager和TrackerDistributedCacheManager三个类。它们的功能如下。

□ DistributedCache类：可供用户直接使用的外部类。它提供了一系列`addXXX`、`setXXX`和`getXXX`方法以配置作业需借用DistributedCache分发的只读文件。

□ TaskDistributedCacheManager类：Hadoop内部使用的类，用于管理一个作业相关的缓存文件。

□ TrackerDistributedCacheManager类：Hadoop内部使用的类，用于管理一个TaskTracker上所有的缓存文件。它只用于缓存`public`可见级别的文件，而对于`private`可见级别的文件，则由`org.apache.hadoop.mapred`包中的`JobLocalizer`类进行缓存。

5.5 小结

作业提交与初始化过程是指从用户输入提交作业命令到作业初始化的整个过程。该过程涉及Hadoop三个非常重要的组件，即JobClient、JobTracker和TaskScheduler。

作业提交主要是为后续作业执行准备环境，涉及创建目录、上传文件等操作。

作业初始化的主要工作是根据输入数据量和作业配置参数将作业分解成若干个Map Task以及Reduce Task，并添加到相关数据结构中，以等待后续被调度执行。

Hadoop DistributedCache是Hadoop为方便用户进行应用程序开发而设计的数据分发工具。它能够将只读的大文件自动分发到各个节点上进行本地缓存，以便Task运行时加载使用。它将待分发的文件根据可见级别分为public级别和private级别两种。其中，public级别文件允许同一个TaskTracker上所有用户共享，而private级别文件只允许某个用户的所有作业共享。

作业在JobTracker端经初始化后，会被存放到相关数据结构中等待被调度执行。在下一章中，我们将重点分析JobTracker和任务调度的相关原理及实现。

第6章 JobTracker内部实现剖析

前面提到，Hadoop MapReduce采用了Master/Slave结构。其中，Master便是这一章将要讲解的JobTracker，它是整个集群中唯一的全局“管理者”，涉及的功能包括作业管理、状态监控、任务调度器等。它的设计思路直接决定着Hadoop MapReduce计算框架的容错性和可扩展性的好坏，因此，它是整个系统中最重要的组件，是系统高效运转的关键。

本章将详细介绍JobTracker的实现细节。总的来看，JobTracker主要包含两个功能：资源管理和作业控制。本章将以这两个功能为切入点深入剖析JobTracker的实现细节，包括状态监控、容错机制、推测执行原理和任务调度机制等。

6.1 JobTracker概述

JobTracker是整个MapReduce计算框架中的主服务，相当于集群的“管理者”，负责整个集群的作业控制和资源管理。在Hadoop内部，每个应用程序被表示成一个作业，每个作业又被进一步分成多个任务，而JobTracker的作业控制模块则负责作业的分解和状态监控。其中，最重要的是状态监控，主要包括TaskTracker状态监控、作业状态监控和任务状态监控等。其主要作用有两个：容错和为任务调度提供决策依据。一方面，通过状态监控，JobTracker能够及时发现存在异常或者出现故障的TaskTracker、作业或者任务，从而启动相应的容错机制进行处理；另一方面，由于JobTracker保存了作业和任务的近似实时运行信息，这些可用于任务调度时进行任务选择的依据。资源管理模块的作用是通过一定的策略将各个节点上的计算资源分配给集群中的任务。它由可插拔的任务调度器完成，用户可根据自己的需要编写相应的调度器。

JobTracker的设计原理如图6-1所示。JobTracker是一个后台服务进程，启动之后，会一直监听并接收来自各个TaskTracker发送的心跳信息，这里面包含节点资源使用情况和任务运行情况等信息。JobTracker会将这些信息统一保存起来，并根据需要为TaskTracker分配新任务。

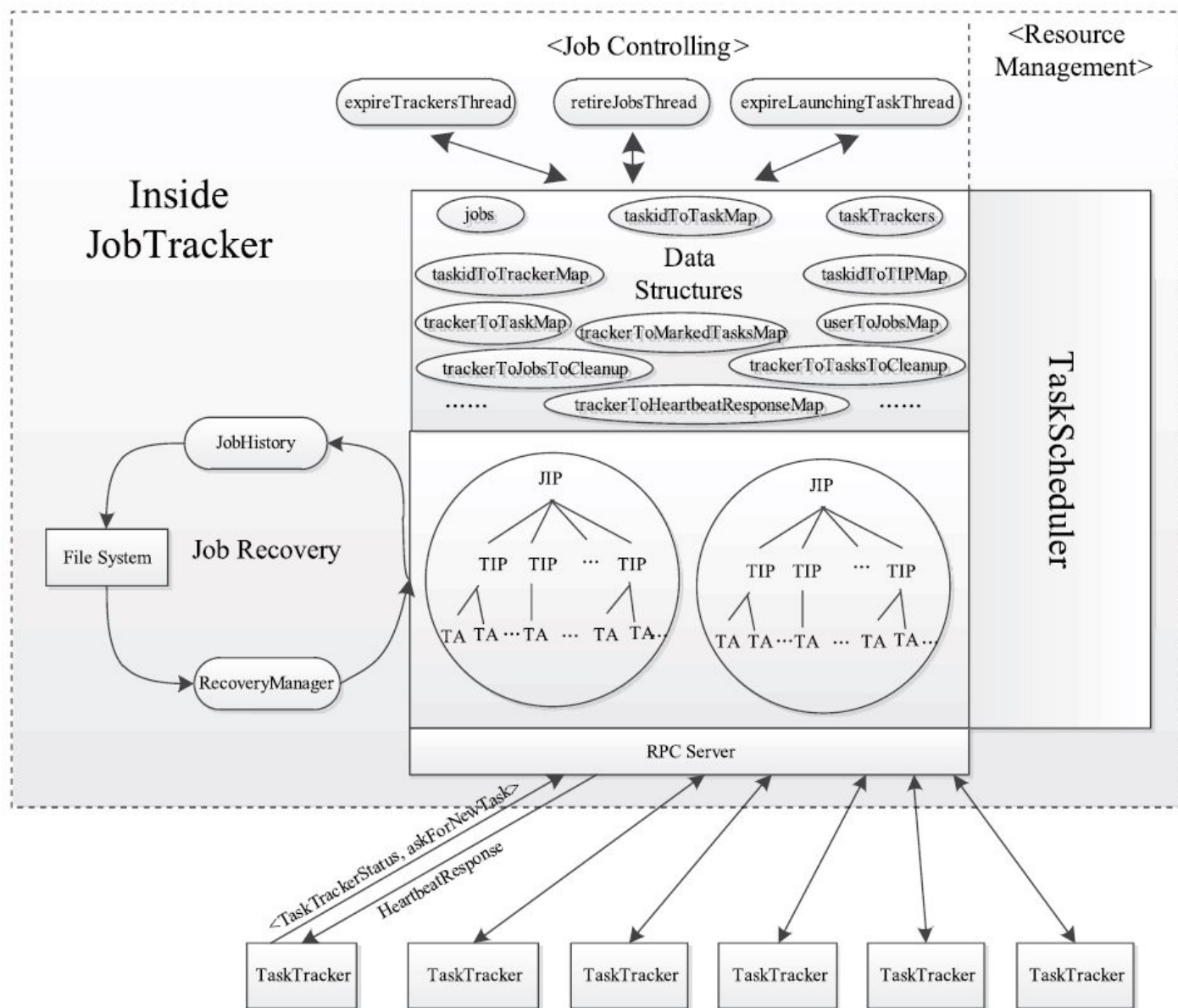


图 6-1 JobTracker内部原理

(1) 作业控制

JobTracker在其内部以“三层多叉树”的方式描述和跟踪每个作业的运行状态，作业被抽象成三层，从上往下依次为：作业监控层、任务监控层和任务执行层。在作业监控层中，每个作业由一个JobInProgress（JIP）对象描述和跟踪其整体运行状态以及每个任务的运行情况；在任务监控层中，每个任务由一个TaskInProgress（TIP）对象描述和跟踪其运行状态；在任务执行层中，考虑到任务在执行过程中可能会失败，因而每个任务可能尝试执行多次，直到成功。JobTracker将每次尝试运行一次任务称为“任务运行尝试”，而对应的任务运行实例称为Task Attempt（TA）。当任何一个Task Attempt运行成功后，其上层对应的TaskInProgress会标注该任务运行成功；而当所有的TaskInProgress运行成功后，JobInProgress则会标注整个作业运行成功。

为了方便查找和定位各种对象（比如TaskTracker，作业或者任务等），JobTracker将其相关信息封装成各种对象后，以key/value的形式保存入数据结构Map中。比如，为了能够根据作业ID找到对应的JobInProgress对象，JobTracker将所有运行作业按照JobID与JobInProgress的对应关系保存入Map^[1]数据结构jobs中；为了能够查找每个TaskTracker上当前正在运行的Task，JobTracker将trackerID与Task ID集合的映射关系保存入Map数据结构trackerToTaskMap中。JobTracker的各种操作，比如监控、更新等，实际上就是修改这些数据结构中的映射关系。

状态监控的一个重要目的是实现容错功能。借助监控信息，JobTracker实现了全方位的容错机制，包括JobTracker、TaskTracker、Job/Task、Record和磁盘等关键服务和对象的容错。此外，通过监控信息，JobTracker可以推测出“拖后腿”的任务，并通过启动备份任务加快数据处理速度。

在Hadoop MapReduce中，JobTracker存在单点故障问题，当失效或者重启后，如果已保存的任务或者节点状态丢失，则所有正在运行的作业将会失败。为了能够在JobTracker发生故障时尽可能大限度地恢复各个作业，Hadoop在作业运行的各个阶段记录日志，以辅助作业恢复。

（2）资源管理

除了状态监控外，JobTracker的另一个重要功能是资源管理。JobTracker不断接收各个TaskTracker周期性发送过来的资源量和任务状态等信息，并综合考虑TaskTracker（所在DataNode）的数据分布、资源剩余量、作业优先级、作业提交时间等因素，为TaskTracker分配最合适的任务。

[1] 在JDK实现中，Map数据结构实际上是红黑树。

6.2 JobTracker启动过程分析

6.2.1 JobTracker启动过程概述

JobTracker是一个后台进程，它包含一个main函数。我们可以从main函数入手，逐步分析JobTracker启动过程。在main函数中有以下两行启动JobTracker的核心代码：

```
JobTracker tracker=startTracker(new JobConf()); //创建JobTracker对象
tracker.offerService(); //启动各个服务
```

本小节主要分析这两行代码的实现细节。其中，函数startTracker()的主要工作是创建一个JobTracker对象，其构造函数的主要工作是对一些重要变量进行初始化；而函数offerService()则是启动JobTracker内部一些重要的服务或者线程。

6.2.2 重要对象初始化

跟踪startTracker()函数内部的执行过程，可定位到它最终创建了一个JobTracker对象。该对象对一些重要对象进行了初始化，具体如表6-1所示。

表 6-1 JobTracker 需初始化的变量列表

对象名	对应类名	意义解释
secretManager	DelegationTokenSecretManager	MapReduce 安全管理相关的类，具体参考第 11 章
aclsManager	ACLsManager	作业级别和队列级别的管理和访问权限控制。作业级别权限包括 VIEW_JOB 和 MODIFY_JOB，而队列级别权限包括 ADMINISTER_JOBS 和 SUBMIT_JOB
taskScheduler	TaskScheduler	调度器对象

(续)

对象名	对应类名	意义解释
interTrackerServer	Server	RPC Server
infoServer	HttpServer	将 Job、Task 和 TaskTracker 相关信息显示到 Web 前端
recoveryManager	RecoveryManager	作业恢复管理，即 JobTracker 启动时，恢复上次停止时正在运行的作业，并恢复各个任务的运行状态
JobHistoryServer	JobHistoryServer	用于查看作业历史信息的 Server
dnsToSwitchMapping	DNSToSwitchMapper	用于构造集群的网络拓扑结构，它能将节点地址（IP 或者 host）映射成网络位置

其中，RPC Server相关内容已在第4章进行了介绍，JobHistoryServer和TaskScheduler将分别在6.2.4节和6.7节介绍，而在本小节中重点分析ACLsManager、HttpServer和DNSToSwitchMapper三个类。

(1) ACLsManager类

它是权限管理类，提供了checkAccess方法以对用户的各种操作进行权限检查。比如，用户提交作业后，JobTracker.submitJob函数中包含以下代码检查用户是否可以提交作业：

```
try{
    aclsManager.checkAccess (job, ugi, Operation.SUBMIT_JOB);
}catch (IOException ioe) {
    LOG.warn ("Access denied for user"+job.getJobConf().getUser()
    +".Ignoring job"+jobId, ioe);
    job.fail();
    throw ioe;
}
```

该类涉及两种权限：队列权限和作业权限，分别由QueueManager类和JobACLsManager类进行管理。

1) QueueManager类：队列权限管理类。在Hadoop中，队列权限包括两部分：作业提交权限（哪些用户可向队列中提交作业）和作业管理权限（哪些用户可以管理该队列中的作业），分别由参数mapred.queue.<queue-name>.acl-submit-job和mapred.queue.<queue-name>.acl-administer-jobs指定，具体在配置文件mapred-queue-acls.xml中设置。

2) JobACLsManager类：作业权限管理类。用户提交作业时，可设定该作业的查看权限和修改权限，分别由参数mapreduce.job.acl-view-job和mapreduce.job.acl-modify-job指定。

作业查看权限主要用于限制访问作业相关的敏感信息，这些信息包括：

❑ 作业级别的Counter。

❑ 任务级别的Counter。

❑ 任务的诊断信息。

❑ TaskTracker的web UI上显示的log信息。

❑ JobTracker的web UI上显示的job.xml文件。

作业修改权限主要用于防止其他用户修改自己作业的信息，这些信息包括：

❑ 杀掉作业。

❑ 杀掉/终止任务。

❑ 修改作业的优先级。

(2) HttpServer类

Hadoop对外提供Web服务的HTTP服务器，它封装了轻量级开源Web服务器Jetty^[1]。

(3) DNSToSwitchMapper接口

该接口定义了将DNS名称或者节点IP地址转换成网络位置的规则。Hadoop以层次树的方式定义节点的网络位置，并依据该位置存取数据或者调度任务。比如，一个节点nodeX在数据中心dcX中的机架rackX上，可以这样表示它的物理位置：/dcX/rackX/nodeX。

默认情况下，Hadoop提供了一个默认实现ScriptBasedMapping，它允许用户通过编写一个脚本（通过参数topology.script.file.name指定）定义转换规则。下面举例说明ScriptBasedMapping的使用方法。

步骤1 用户将节点与网络位置映射关系放到目录\${HADOOP_HOME}/conf下的topology.data文件中，形式如下：

```
node1/dc1/rack1
node2/dc1/rack1
node3/dc1/rack1
node4/dc2/rack2
```

步骤2 编写Shell脚本node2rack.sh，内容如下：

```
#!/bin/bash
HADOOP_CONF=${HADOOP_HOME}/conf
while[ $# -gt 0 ]; do
  nodeArg=$1
  exec< ${HADOOP_CONF}/topology.data
  result=""
  while read line; do
    ar=($line)
    if["${ar[0]}"=="$nodeArg"]; then
      result="${ar[1]}"
    fi
  done
  shift
  if[-z "$result"]; then
    echo -n "/default/rack"
  else
    echo -n "$result"
  fi
done
```

步骤3 在core-site.xml中添加配置选项，具体如下：

```
<property>
<name>topology.script.file.name</name>
<value>/opt/scripts/node2rack.sh</value>
</property>
```

当然，用户也可以实现DNSToSwitchMapper接口，并通过配置参数`topology.node.switch.mapping.impl`指定对应的实现类。

[1] <http://jetty.codehaus.org/jetty/>

6.2.3 各种线程功能

函数offerServer会启动JobTracker内部几个比较重要的后台服务线程，分别是expireTrackersThread、retireJobsThread、expireLaunchingTaskThread和completedJobsStoreThread。下面分别介绍这几个服务线程。

（1）expireTrackersThread线程

该线程主要用于发现和清理死掉的TaskTracker。每个TaskTracker会周期性地通过心跳向JobTracker汇报信息，而JobTracker会记录每个TaskTracker最近的汇报心跳时间。如果某个TaskTracker在10分钟内未汇报心跳，则JobTracker认为它已死掉，并将它的相关信息从数据结构trackerToJobsToCleanup、trackerToTasksToCleanup、trackerToTaskMap、trackerToMarkedTasksMap中清除，同时将正在运行的任务状态标注为KILLED_UNCLEAN。

（2）retireJobsThread线程

该线程主要用于清理长时间驻留在内存中的已经运行完成的作业信息。JobTracker会将已经运行完成的作业信息存放到内存中，以便外部查询，但随着完成的作业越来越多，势必会占用JobTracker的大量内存，为此，JobTracker通过该线程清理驻留在内存中较长时间的已经运行完成的作业信息。

当一个作业满足如下条件1、2或者条件1、3时，将被从数据结构jobs转移到过期作业队列中。

条件1 作业已经运行完成，即运行状态为SUCCEEDED、FAILED或KILLED。

条件2 作业完成时间距现在已经超过24小时（可通过参数mapred.jobtracker.retirejob.interval配置）。

条件3 作业拥有者已经完成作业总数超过100（可通过参数mapred.jobtracker.completeuserjobs.maximum配置）个。

过期作业被统一保存到过期队列中。当过期作业超过1 000个（可通过参数mapred.job.tracker.retiredjobs.cache.size配置）时，将会从内存中彻底删除。

（3）expireLaunchingTaskThread线程

该线程用于发现已经被分配给某个TaskTracker但一直未汇报信息的任务。当JobTracker将某个任务分配给TaskTracker后，如果该任务在10分钟内未汇报进度，则JobTracker认为该任务分配失败^[1]，并将其状态标注为FAILED。

（4）completedJobsStoreThread线程

该线程将已经运行完成的作业运行信息保存到HDFS上，并提供了一套存取这些信息的API。该线程能够解决以下两个问题。

□用户无法获取很久之前的作业运行信息：前面提到线程retireJobsThread会清除长时间驻留在内存中的完成作业，这会导致用户无法查询很久之前某个作业的运行信息。

□JobTracker重启后作业运行信息丢失：当JobTracker因故障重启后，所有原本保存到内存中的作业信息将会全部丢失。

该线程通过保存作业运行日志的方式，使得用户可以查询任意时间提交的作业和还原作业的运行信息。

默认情况下，该线程不会启用，用户可通过表6-2所示的几个参数配置并启用该线程。

表 6-2 completedJobsStoreThread 线程控制参数

配置参数	参数含义
mapred.job.tracker.persist.jobstatus.active	是否启用该线程
mapred.job.tracker.persist.jobstatus.hours	作业运行信息保存时间
mapred.job.tracker.persist.jobstatus.dir	作业运行信息保存路径

[1] JobTracker总是假设TaskTracker是不可靠的，它总是认为TaskTracker可能会只接收新任务但不启动它。

6.2.4 作业恢复

在MapReduce中，JobTracker存在单点故障问题。如果它因异常退出后重启，那么所有正在运行的作业运行时信息将丢失。如果不采用适当的作业恢复机制对作业信息进行恢复，则所有作业需重新提交，且已经计算完成的任务需重新计算。这势必造成资源浪费。

为了解决JobTracker面临的单点故障问题，Hadoop设计了作业恢复机制，过程如下：作业从提交到运行结束的整个过程中，JobTracker会为一些关键事件记录日志（由JobHistory类完成）。对于作业而言，关键事件包括作业提交、作业创建、作业开始运行、作业运行完成、作业运行失败、作业被杀死等；对于任务而言，关键事件包括任务创建、任务开始运行、任务运行结束、任务运行失败、任务被杀死等。当JobTracker因故障重启后（重启过程中，所有TaskTracker仍然活着），如果管理员启用了作业恢复功能（将参数mapred.jobtracker.restart.recover置为true），则JobTracker会检查是否存在需要恢复运行状态的作业，如果有，则通过日志恢复这些作业的运行状态（由RecoveryManager类完成），并重新调度那些未运行完成的任务（包括产生部分结果的任务）。

6.3 心跳接收与应答

心跳是沟通TaskTracker与JobTracker的桥梁，它实际上是一个RPC函数。TaskTracker周期性地调用该函数汇报节点和任务状态信息，从而形成心跳。在Hadoop中，心跳主要有三个作用：

- ❑判断TaskTracker是否活着。
- ❑及时让JobTracker获取各个节点上的资源使用情况和任务运行状态。
- ❑为TaskTracker分配任务。

注意 JobTracker与TaskTracker之间采用了“pull”而不是“push”模型，即JobTracker从不会主动向TaskTracker发送任何信息，而是由TaskTracker主动通过心跳“领取”属于自己的信息。JobTracker只能通过心跳应答的形式为各个TaskTracker分配任务。

TaskTracker周期性地调用RPC函数heartbeat向JobTracker汇报信息和领取任务。该函数定义如下：

```
public synchronized HeartbeatResponse heartbeat (TaskTrackerStatus status,
boolean restarted,
boolean initialContact,
boolean acceptNewTasks,
short responseId)
```

该函数的各个参数含义如下。

❑Status：该参数封装了TaskTracker上的各种状态信息，包括

```
String trackerName; //TaskTracker名称，形式如tracker_mymachine: localhost.
localdomain/127.0.0.1: 34196
String host; //TaskTracker主机名
int httpPort; //TaskTracker对外的HTTP端口号
int failures; //该TaskTracker上已经失败的任务总数
List<TaskStatus>taskReports; //正在运行的各个任务运行状态
volatile long lastSeen; //上次汇报心跳的时间
private int maxMapTasks; /*Map slot总数，即允许同时运行的Map Task总数，由参数mapred.
tasktracker.map.tasks.maximum设定*/
private int maxReduceTasks; //Reduce slot总数
private TaskTrackerHealthStatus healthStatus; //TaskTracker健康状态
private ResourceStatus resStatus; //TaskTracker资源（内存，CPU等）信息
```

❑Restarted：表示TaskTracker是否刚刚重新启动。

❑initialContact：表示TaskTracker是否初次连接JobTracker。

❑acceptNewTasks：表示TaskTracker是否可以接收新任务，这通常取决于slot是否有剩余和节点健康状况等。

❑responseId：表示心跳响应编号，用于防止重复发送心跳。每接收一次心跳后，该值加1。

该函数的返回值为一个HeartbeatResponse对象，该对象主要封装了JobTracker向TaskTracker下达的命令，具体如下：

```
class HeartbeatResponse implements Writable, Configurable{
.....
short responseId; //心跳响应编号
int heartbeatInterval; //下次心跳的发送间隔
TaskTrackerAction[]actions; /*来自JobTracker的命令，可能包括杀死作业、杀死任务、提
交任务、运行任务等，具体参考6.3.2节*/
Set<JobID>recoveredJobs=new HashSet<JobID> (); //恢复完成的作业列表，具体参考
6.2.4节
.....
}
```

该函数的内部实现逻辑主要分为两个步骤：更新状态和下达命令。JobTracker首先将TaskTracker汇报的最新任务运行状态保存到相应数据结构中，然后根据这些状态信息和外界需求（比如用户杀死一个作业）为其下达相应的命令。

6.3.1 更新状态

函数heartbeat首先会更新TaskTracker/Job/Task的状态信息。相关代码如下：

```
/*检查是否允许该TaskTracker连接JobTracker。当一个TaskTracker在host list（由参数mapred.hosts指定）中，但不在exclude list（由参数mapred.hosts.exclude指定）中时，可接入JobTracker*/
if (! acceptTaskTracker (status)) {
    throw new DisallowedTaskTrackerException (status);
}
.....
/*如果该TaskTracker被重启了，则将之标注为健康的TaskTracker，并从黑名单或者灰名单中清除（关于黑名单与灰名单的介绍，参考6.5.2节），否则，启动TaskTracker容错机制以检查它是否处于健康状态*/
if (restarted) {
    faultyTrackers.markTrackerHealthy (status.getHost());
} else {
    faultyTrackers.checkTrackerFaultTimeout (status.getHost(), now);
}
.....
short newResponseId= (short) (responseId+1); //响应编号加1
/*记录心跳发送时间，以发现在一定时间内未发送心跳的TaskTracker，并将之标注为死亡的TaskTracker，此后不可再向其分配新任务*/
status.setLastSeen (now);
if (! processHeartbeat (status, initialContact, now)) { //处理心跳
.....
```

接下来，跟踪进入函数processHeartbeat内部。该函数首先进行一系列异常情况检查，然后调用以下两个函数更新TaskTracker/Job/Task的状态信息：

```
updateTaskStatuses (trackerStatus); //更新Task状态信息
updateNodeHealthStatus (trackerStatus, timeStamp); //更新节点健康状态
```

6.3.2 下达命令

更新完状态信息后，JobTracker要为TaskTracker构造一个HeartbeatResponse对象作为心跳应答。该对象主要有两部分内容：下达给TaskTracker的命令和下次汇报心跳的时间间隔。下面分别对它们进行介绍。

1. 下达命令

JobTracker将下达给TaskTracker的命令封装成TaskTrackerAction类，主要包括ReinitTrackerAction（重新初始化）、LaunchTaskAction（运行新任务）、KillTaskAction（杀死任务）、KillJobAction（杀死作业）和CommitTaskAction（提交任务）五种。下面依次对这几个命令进行介绍。

（1）ReinitTrackerAction

JobTracker收到TaskTracker发送过来的心跳信息后，首先要进行一致性检查，如果发现异常情况，则会要求TaskTracker重新对自己进行初始化，以恢复到一致的状态。当出现以下两种不一致情况时，JobTracker会向TaskTracker下达ReinitTrackerAction命令。

❑ 丢失上次心跳应答信息：JobTracker会保存向每个TaskTracker发送的最近心跳应答信息，如果JobTracker未刚刚重启且一个TaskTracker并非初次连接JobTracker（initialContact! =true），而最近的心跳应答信息丢失了，则这是一种不一致状态。

❑ 丢失TaskTracker状态信息：JobTracker接收到任何一个心跳信息后，会将TaskTracker状态（封装在类TaskTrackerStatus中）信息保存起来。如果一个TaskTracker非初次连接JobTracker但状态信息却不存在，则也是一种不一致状态。

（2）LaunchTaskAction

该类封装了TaskTracker新分配的任务。TaskTracker接收到该命令后会启动一个子进程运行该任务。Hadoop将一个作业分解后的任务分成两大类：计算型任务和辅助型任务。其中，计算型任务是处理实际数据的任务，包括Map Task和Reduce Task两种（对应TaskType类中的MAP和REDUCE两种类型），由专门的任务调度器对它们进行调度；而辅助型任务则不会处理实际的数据，通常用于同步计算型任务或者清理磁盘上无用的目录，包括job-setup task、job-cleanup task和task-cleanup task三种（对应TaskType类中的JOB_SETUP, JOB_CLEANUP和TASK_CLEANUP三种类型），其中，job-setup task和job-cleanup task分别用作计算型任务开始运行同步标识和结束运行同步标识，其具体特点已在第4章进行了介绍，而task-cleanup task则用于清理失败的计算型任务已经写到磁盘上的部分结果，这种任务由JobTracker负责调度，且运行优先级高于计算型任务。

如果一个正常（不在黑名单中）的TaskTracker尚有空闲slot（acceptNewTasks为true），则JobTracker会为该TaskTracker分配新任务，任务选择顺序是：先辅助型任务，再计算型任务。而对于辅助型任务，选择顺序依次为job-cleanup task、task-cleanup task和job-setup task，具体代码如下：

```
/*优先选择辅助型任务，选择优先级从高到低依次为：job-cleanup task、task-cleanup task和
job-setup task，这样可以让运行完成的作业快速结束，新提交的作业立刻进入运行状态*/
List<Task> tasks=getSetupAndCleanupTasks (taskTrackerStatus);
//如果没有辅助型任务，则选择计算型任务
if (tasks==null) {
    //由任务调度器选择一个或多个计算型任务
    tasks=taskScheduler.assignTasks (taskTrackers.get (trackerName));
}
if (tasks!=null) {
    for (Task task: tasks) {
        expireLaunchingTasks.addNewTask (task.getTaskID());
        //将分配的任务封装成LaunchTaskAction对象
        actions.add (new LaunchTaskAction (task));
    }
}
```

（3）KillTaskAction

该类封装了TaskTracker需杀死的任务。TaskTracker收到该命令后会杀掉对应任务、清理工作目录和释放slot。导致JobTracker向TaskTracker发送该命令的原因有很多，主要包括以下几个场景：

- ❑ 用户使用命令“bin/hadoop job-kill-task”或者“bin/hadoop job-fail-task”杀死一个任务或者使一个任务失败。
- ❑ 启用推测执行机制后，同一份数据可能同时由两个Task Attempt处理。当其中一个Task Attempt执行成功后，另外一个处理相同数据的Task Attempt将被杀掉。
- ❑ 某个作业运行失败，它的所有任务将被杀掉。
- ❑ TaskTracker在一定时间内未汇报心跳，则JobTracker认为其死掉，它上面的所有Task均被标注为死亡。

（4）KillJobAction

该类封装了TaskTracker待清理的作业。TaskTracker接收到该命令后，会清理作业的临时目录。导致JobTracker向TaskTracker发送该命令的原因有很多，主要包括以下几个场景：

- ❑ 用户使用命令“bin/hadoop job-kill”或者“bin/hadoop job-fail”杀死一个作业或者使一个作业失败。
- ❑ 作业运行完成，通知TaskTracker清理该作业的工作目录。
- ❑ 作业运行失败，即同一个作业失败的Task数目超过一定比例。

（5）CommitTaskAction

该类封装了TaskTracker需提交的任务。为了防止同一个TaskInProgress的两个同时运行的Task Attempt（比如打开推测执行功能，一个任务可能存在备份任务）同时打开一个文件或者往一个文件中写数据而产生冲突，Hadoop让每个Task Attempt写到单独一个文件（以TaskAttemptID命名，比如attempt_201208071706_0008_r_000000_0）中。通常而言，Hadoop让每个Task Attempt将计算结果写到临时目录\${mapred.output.dir}/_temporary/_\${taskid}中，当某个Task Attempt成功运行完成后，再将运算结果转移到最终目录\${mapred.output.dir}中。Hadoop将一个成功运行完成的Task Attempt结果文件从临时目录“提升”至最终目录的过程，称为“任务提交”。当TaskInProgress中一个任务被提交后，其他任务将被杀死，同时意味着该TaskInProgress运行完成。

2.调整心跳间隔

TaskTracker心跳时间间隔大小应该适度，如果太小，则JobTracker需要处理高并发的心跳连接请求，必然产生不小的并发压力；如果太大，空闲的资源不能及时汇报给JobTracker（进而为之分配新的Task），造成资源空闲，进而降低系统吞吐率。

TaskTracker汇报心跳的时间间隔并不是一成不变的，它会随着集群规模的动态调整（比如节点死掉或者用户动态添加新节点）而变化，以便能够合理利用JobTracker的并发处理能力。在Hadoop MapReduce中，只有JobTracker知道某一时刻集群的规模，因此由JobTracker为每个TaskTracker计算下一次汇报心跳的时间间隔，并通过心跳机制告诉TaskTracker。

JobTracker允许用户通过参数配置心跳的时间间隔加速比，即每增加mapred.heartbeats.in.second（默认是100，最小是1）个节点，心跳时间间隔增加mapreduce.jobtracker.heartbeats.scaling.factor（默认是1，最小是0.01）秒。同时，为了防止用户参数设置不合理而对JobTracker产生较大负载，JobTracker要求心跳时间间隔至少为3秒^[1]。具体计算方法如下：

```
public int getNextHeartbeatInterval(){
//获取当前TaskTracker总数，即集群当前规模
int clusterSize=getClusterStatus().getTaskTrackers();
//计算新的心跳间隔
int heartbeatInterval=Math.max(
(int)(1000*HEARTBEATS_SCALING_FACTOR*
Math.ceil((double)clusterSize/
```

```
NUM_HEARTBEATS_IN_SECOND) ),
HEARTBEAT_INTERVAL_MIN);
return heartbeatInterval;
}
```

[1] 考虑在中小规模集群下，心跳时间间隔至少为3秒，这会降低系统吞吐率，在最新版本中，已经调整为300毫秒，具体参考：
<https://issues.apache.org/jira/browse/MAPREDUCE-1906>。

6.4 Job和Task运行时信息维护

JobTracker最重要的功能之一是状态监控，包括TaskTracker、Job和Task等运行时状态的监控，其中，TaskTracker状态监控比较简单，只要记录其最近心跳汇报时间和健康状况（由TaskTracker端的监控脚本检测，并通过心跳将结果发送给JobTracker）即可。本节重点分析Job和Task的监控方法，内容涉及作业描述模型、作业/任务运行时监控信息、作业/任务状态转换等。

6.4.1 作业描述模型

如图6-2所示，JobTracker在其内部以“三层多叉树”的方式描述和跟踪每个作业的运行状态。JobTracker为每个作业创建一个JobInProgress对象以跟踪和监控其运行状态。该对象存在于作业的整个运行过程中：它在作业提交时创建，作业运行完成时销毁。同时，为了采用分而治之的策略解决问题，JobTracker会将每个作业拆分成若干个任务，并为每个任务创建一个TaskInProgress对象以跟踪和监控其运行状态，而任务在运行过程中，可能会因为软件Bug、硬件故障等原因运行失败，此时JobTracker会按照一定的策略重新运行该任务，也就是说，每个任务可能会尝试运行多次，直到运行成功或者因超过尝试次数而失败。JobTracker将每运行一次任务称为一次“任务运行尝试”，即Task Attempt。对于某个任务，只要有一个Task Attempt运行成功，则相应的TaskInProgress对象会标注该任务运行成功，而当所有的TaskInProgress均标注其对应的任务运行成功后，JobInProgress对象会标识整个作业运行成功。

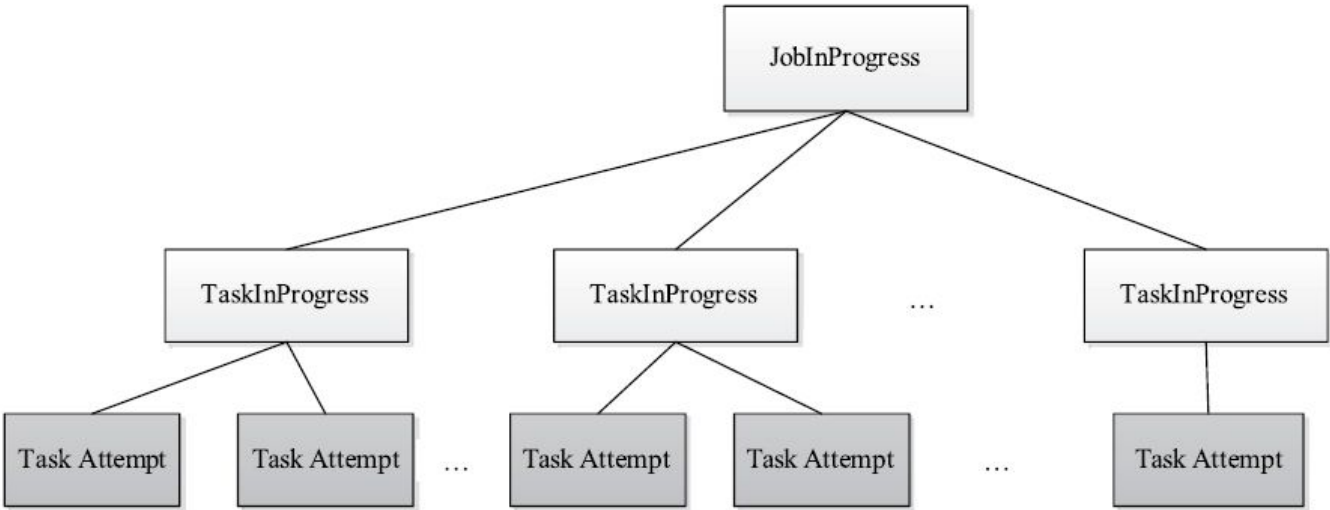


图 6-2“三层多叉树”作业描述方式

如图6-3所示，为了区分各个作业，JobTracker会赋予每个作业一个唯一的ID。该ID由三部分组成：作业前缀字符串、JobTracker启动时间和作业提交顺序，各部分通过“_”连接起来组成一个完整的作业ID，比如job_201208071706_0009，对应的三部分分别是“job”、“201208071706”和“009”（JobTracker运行以来第9个作业）。每个任务的ID继承了作业的ID，并在此基础上进行了扩展，它由三部分组成：作业ID（其中前缀字符串变为“task”）、任务类型（map还是reduce）和任务编号（从000000开始，一直到999999）。比如，task_201208071706_0009_m_000000，表示它的作业ID为task_201208071706_0009，任务类型为map，任务编号为000000。每个Task Attempt的ID继承了任务的ID，它由两部分组成：任务ID（其中前缀字符串变为“attempt”）和运行尝试次数（从0开始），比如，attempt_201208071706_0009_m_000000_0表示任务task_201208071706_0009_m_000000的第0次尝试。

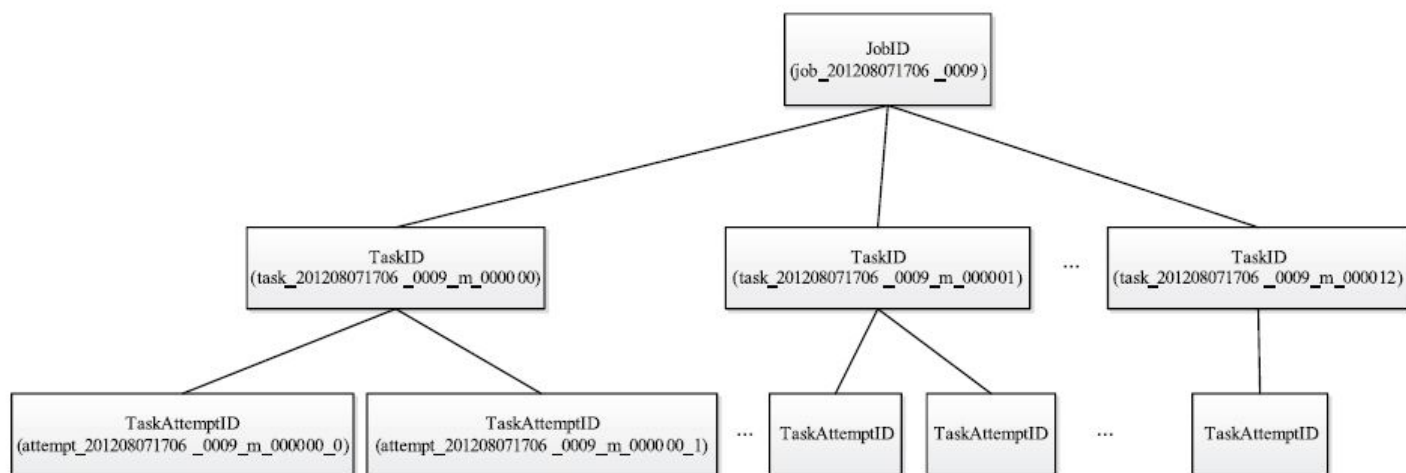


图 6-3 Job/Task/TaskAttempt的ID继承关系

6.4.2 JobInProgress

JobInProgress类主要用于监控和跟踪作业运行状态，并为调度器提供最底层的调度接口。本小节主要介绍其作业监控信息，而其调度函数相关实现将在6.7节中介绍。

JobInProgress维护了两种作业信息：一种是静态信息，这些信息是作业提交之时就已经确定好的；另一种是动态信息，这些信息随着作业的运行而动态变化。

(1) 作业静态信息

作业静态信息是指作业提交之时就已经确定好的属性信息，主要包括以下几项：

```
//map task, reduce task, cleanup task和setup task对应的TaskInProgress
TaskInProgress maps[]=new TaskInProgress[0];
TaskInProgress reduces[]=new TaskInProgress[0];
TaskInProgress cleanup[]=new TaskInProgress[0];
TaskInProgress setup[]=new TaskInProgress[0];
int numMapTasks=0; //Map Task个数
int numReduceTasks=0; //Reduce Task个数
final long memoryPerMap; //每个Map Task需要的内存量
final long memoryPerReduce; //每个Reduce Task需要的内存量
volatile int numSlotsPerMap=1; //每个Map Task需要的slot个数
volatile int numSlotsPerReduce=1; //每个Reduce Task需要的slot个数
/*允许每个TaskTracker上失败的Task个数，默认是4，通过参数mapred.max.tracker.failures
设置。当该作业在某个TaskTracker上失败的个数超过该值时，会将该节点添加到该作业的黑名单中，调度
器便不再为该节点分配该作业的任务*/
final int maxTaskFailuresPerTracker;
.....
private static float DEFAULT_COMPLETED_MAPS_PERCENT_FOR_REDUCE_SLOWSTART=0.05f;
//当有5%的Map Task完成后，才可以调度Reduce Task
int completedMapsForReduceSlowstart=0; //多少Map Task完成后开始调度Reduce Task
.....
//允许的Map Task失败比例上限，通过参数mapred.max.map.failures.percent设置
final int mapFailuresPercent;
//允许的Reduce Task失败比例上限，通过参数mapred.max.reduce.failures.percent设置
final int reduceFailuresPercent;
.....
JobPriority priority=JobPriority.NORMAL; //作业优先级
```

(2) 作业动态信息

作业动态信息是指作业运行过程中会动态更新的信息。这些信息对于发现TaskTracker/Job/Task故障非常有用，也可以为调度器进行任务调度提供决策依据。

```
int runningMapTasks=0; //正在运行的Map Task数目
int runningReduceTasks=0; //正在运行的Reduce Task数目
int finishedMapTasks=0; //运行完成的Map Task数目
int finishedReduceTasks=0; //运行完成的Reduce Task数目
int failedMapTasks=0; //失败的Map Task Attempt数目
int failedReduceTasks=0; //失败的Reduce Task Attempt数目
.....
int speculativeMapTasks=0; //正在运行的备份任务（MAP）数目
int speculativeReduceTasks=0; //正在运行的备份任务（REDUCE）数目
int failedMapTIPs=0; /*失败的TaskInProgress（MAP）数目，这意味着对应的输入数据将被丢弃，
不会产生最终结果*/
int failedReduceTIPs=0; //失败的TaskInProgress（REDUCE）数目
private volatile boolean launchedCleanup=false; //是否已启动Cleanup Task
private volatile boolean launchedSetup=false; //是否已启动Setup Task
private volatile boolean jobKilled=false; //作业是否已被杀死
private volatile boolean jobFailed=false; //作业是否已失败
//节点与TaskInProgress的映射关系，即TaskInProgress输入数据位置与节点对应关系
Map<Node, List<TaskInProgress>>nonRunningMapCache;
//节点及其上面正在运行的Task映射关系
Map<Node, Set<TaskInProgress>>runningMapCache;
/*不需要考虑数据本地性的Map Task，如果一个Map Task的InputSplit Location为空，则进行任
务调度时不需考虑本地性*/
final List<TaskInProgress>nonLocalMaps;
//按照失败次数进行排序的TIP集合
final SortedSet<TaskInProgress>failedMaps;
//未运行的Map Task集合
Set<TaskInProgress>nonLocalRunningMaps;
//未运行的Reduce Task集合
Set<TaskInProgress>nonRunningReduces;
//正在运行的Reduce Task集合
Set<TaskInProgress>runningReduces;
//待清理的Map Task列表，比如用户直接通过命令“bin/hadoop job-kill”杀死的Task
```

```
List<TaskAttemptID>mapCleanupTasks=new LinkedList<TaskAttemptID>();
List<TaskAttemptID>reduceCleanupTasks=new LinkedList<TaskAttemptID>();
long startTime; //作业提交时间
long launchTime; //作业开始执行时间
long finishTime; //作业完成时间
```

6.4.3 TaskInProgress

TaskInProgress类维护了一个Task运行过程中的全部信息。在Hadoop中，由于一个任务可能会推测执行或者重新执行，所以会存在多个Task Attempt，且同一时刻，可能有多个处理相同的任务尝试同时在执行，而这些任务被同一个TaskInProgress对象管理和跟踪，只要任何一个任务尝试运行成功，TaskInProgress就会标注该任务执行成功。

```
private final TaskSplitMetaInfo splitInfo; //Task要处理的Split信息
private int numMaps; //Map Task数目, 只对Reduce Task有用
private int partition; //该Task在task列表中的索引
private JobTracker jobtracker; //JobTracker对象, 用于获取全局时钟
private TaskID id; //task ID, 其后面加下标构成Task Attempt ID
private JobInProgress job; //该TaskInProgress所在的JobInProgress
private final int numSlotsRequired; //运行该Task需要的slot数目
private int successEventNumber=-1;
private int numTaskFailures=0; //Task Attempt失败次数
private int numKilledTasks=0; //Task Attempt被杀死次数
private double progress=0; //任务运行进度
private String state=""; //运行状态
private long startTime=0; //TaskInProgress对象创建时间
private long execStartTime=0; //第一个Task Attempt开始运行时间
private long execFinishTime=0; //最后一个运行成功的Task Attempt完成时间
private int completes=0; //Task Attempt运行完成数目, 实际只有两个值: 0和1
private boolean failed=false; //该TaskInProgress是否运行失败
private boolean killed=false; //该TaskInProgress是否被杀死
private boolean jobCleanup=false; //该TaskInProgress是否为Cleanup Task
private boolean jobSetup=false; //该TaskInProgress是否为Setup Task
//该TaskInProgress的下一个可用Task Attempt ID
int nextTaskId=0;
//使得该TaskInProgress运行成功的那个Task ID
private TaskAttemptID successfulTaskId;
//第一个运行的Task Attempt的ID
private TaskAttemptID firstTaskId;
//正在运行的Task ID与TaskTracker ID之间的映射关系
private TreeMap<TaskAttemptID, String>activeTasks=new TreeMap<TaskAttemptID, String>();
//该TaskInProgress已运行的所有TaskAttempt ID, 包括已经运行完成的和正在运行的
private TreeSet<TaskAttemptID>tasks=new TreeSet<TaskAttemptID>();
//Task ID与TaskStatus映射关系
private TreeMap<TaskAttemptID, TaskStatus>taskStatuses=
new TreeMap<TaskAttemptID, TaskStatus>();
//Cleanup Task ID与TaskTracker ID映射关系
private TreeMap<TaskAttemptID, String>cleanupTasks=
new TreeMap<TaskAttemptID, String>();
//所有已经运行失败的Task所在的节点列表
private TreeSet<String>machinesWhereFailed=new TreeSet<String>();
//某个Task Attempt运行成功后, 其他所有正在运行的Task Attempt保存在该集合中
private TreeSet<TaskAttemptID>tasksReportedClosed=new TreeSet<TaskAttemptID>();
//待杀死的Task列表
private TreeMap<TaskAttemptID, Boolean>tasksToKill=new TreeMap<TaskAttemptID,
Boolean>();
//等待被提交的Task Attempt, 该Task Attempt最终使得TaskInProgress运行成功
private TaskAttemptID taskToCommit;
```

6.4.4 作业和任务状态转换图

在Hadoop MapReduce中，作业和任务是有生命周期的，它们的状态受各种行为的影响而发生变化。在本小节中，我们将分析作业和任务涉及的状态转移以及导致状态转移的事件。

(1) 作业状态转换图

前面提到，作业的运行时信息由JobInProgress类进行监控和维护，因此，作业状态转移也由该类进行更新。在Hadoop中，一个作业在运行过程中可能涉及所有可能的状态转移，如图6-4所示。

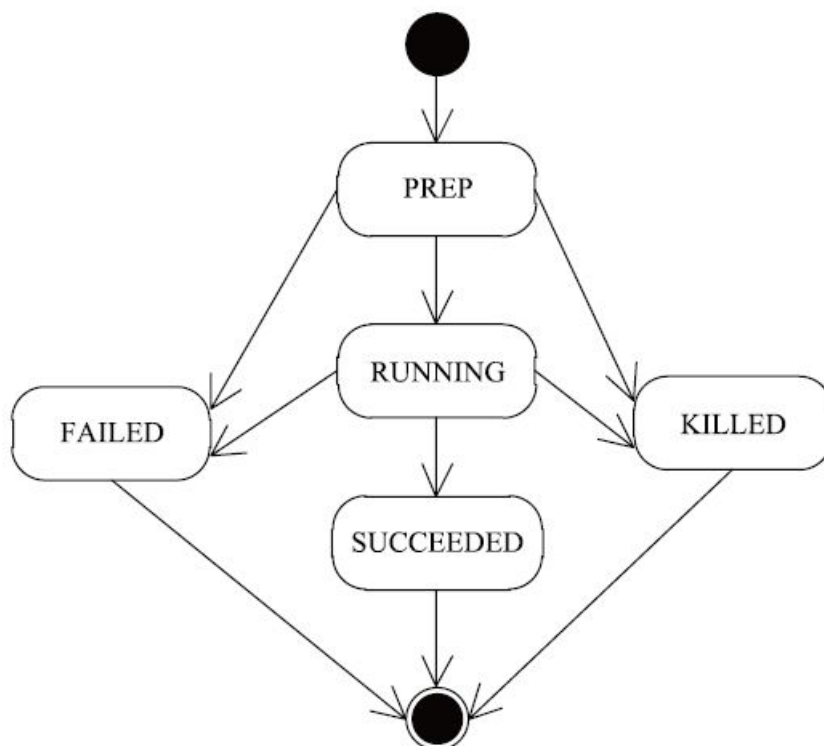


图 6-4 作业状态转换图

图6-4中涉及的状态转换以及对应事件如下。

□ PREP→RUNNING: 作业的Setup Task (job-setup Task) 成功执行完成。

□ RUNNING→SUCCEEDED: 作业的Cleanup Task (job-cleanup Task) 执行成功。

□ PREP→FAILED/KILLED: 人为使用Shell命令杀死作业，即bin/hadoop job[-kill|-fail]<jobid>。

□ RUNNING→FAILED: 多种情况可导致该状态转移，包括人为使用Shell命令杀死作业（使用“bin/hadoop job-fail<jobid>”命令），作业的Cleanup/Setup Task运行失败和作业失败的任务数超过了一定比例。

□ RUNNING→KILLED: 人为使用Shell命令杀死作业，比如使用“bin/hadoop job-kill<jobid>”命令。

(2) 任务状态转换图

在Hadoop中，一个任务的状态变化可能发生在JobTracker端或者TaskTracker端。总结起来，一个任务在运行过程中可能涉及所有可能的状态转移，如图6-5所示。

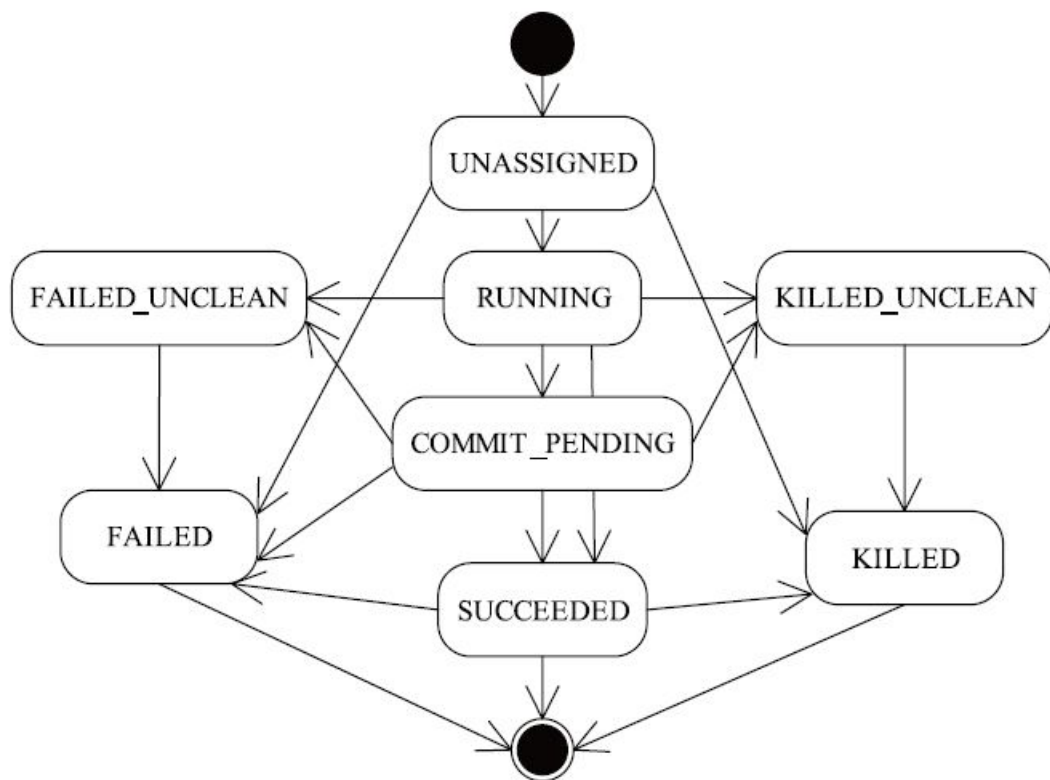


图 6-5 任务状态转换图

图6-5中涉及的状态转换以及对应事件如下。

□UNASSIGNED→RUNNING: 任务初始化状态为UNASSIGNED，当JobTracker将任务分配给某个TaskTracker后，该TaskTracker会为它准备运行环境并启动它，之后该任务进入RUNNING状态。

□RUNNING→COMMIT_PENDING: 该状态转换存在于产生最终结果的任务（Reduce Task或者map-only类型作业^[1]的Map Task）中，当任务处理完最后一条记录后进入COMMIT_PENDING状态，以等待JobTracker批准其提交最后结果。

□RUNNING→SUCCEEDED:该状态转换只存在于Map Task（且这些Map Task的结果将被Reduce Task进一步处理）中，当Map Task处理完最后一条记录后便意味着任务运行成功。

□RUNNING/COMMIT_PENDING→KILLED_UNCLEAN: TaskTracker收到来自JobTracker的KillTaskAction命令后，会将对应任务由RUNNING/COMMIT_PENDING状态转化为KILLED_UNCLEAN状态，通常产生的场景是人为杀死任务，同一个TIP的多个同时运行的Task Attempt中有一个成功运行完成而杀死其他Task Attempt, TaskTracker因超时导致其上所有任务状态变为KILLED_UNCLEAN等。

□RUNNING/COMMIT_PENDING→FAILED_UNCLEAN: 多种情况下会导致该状态转移，包括本地文件读写错误、Shuffle阶段错误、任务在一定时间内未汇报进度（从而被TaskTracker杀掉）、内存使用量超过期望值或者其他运行过程中出现的错误。

□UNASSIGNED→FAILED/KILLED:人为杀死任务。

□KILLED_UNCLEAN/FAILED_UNCLEAN→FAILED/KILLED:一旦任务进入KILLED_UNCLEAN/FAILED_UNCLEAN状态，接下来必然进入FAILED/KILLED状态，以清理已经写入HDFS上的部分结果。

□SUCCEEDED→KILLED:一个TIP已有一个Task Attempt运行完成，而备份任务也汇报成功，则备份任务将被杀掉或者用户人为杀死某个Task，而TaskTracker刚好汇报对应Task执行成功。

□SUCCEEDED/COMMIT_PENDING→FAILED:Reduce Task从Map Task端远程读取数据时，发现数据损坏或者丢失，则将对

应Map Task状态标注为FAILED以便重新得到调度。

[1] Map-only类型作业是指只有Map Task而没有Reduce Task的作业。对于这种作业，Map Task直接将结果写到HDFS上。

6.5 容错机制

6.5.1 JobTracker容错

在MapReduce中，JobTracker掌握着整个集群的运行信息，包括节点健康状况、资源分布情况以及所有作业的运行信息。如果JobTracker因故障而重启，部分信息很容易通过心跳机制重新构造，比如节点健康情况和资源分布情况；但对于作业的运行信息而言，可能将会全部丢失，这使得用户不得不重新提交未运行完成的作业，这意味着之前已经运行完成的任务不得不重新运行，进而造成资源浪费。从以上分析可看出，JobTracker容错的关键技术点是如何保存和恢复作业的运行信息。

从作业恢复粒度角度看，当前存在三种不同级别的恢复机制，级别由低到高依次是作业级别、任务级别和记录级别，其中，级别越低，实现越简单，但造成的资源浪费越严重。在1.0.0以及之前版本中，Hadoop采用了任务级别的恢复机制^[1]，即以任务为基本单位进行恢复，这种机制是基于事务型日志完成作业恢复的，它只关注两种任务：运行完成的任务和未运行完成的任务。作业执行过程中，JobTracker会以日志的形式将作业以及任务状态记录下来，一旦JobTracker重启，则可从日志中恢复作业的运行状态，其中已经运行完成的任务无须再运行，而未开始运行或者运行中的任务需重新运行。这种方案实现比较复杂，需要处理的特殊情况比较多，为了简化设计，从0.21.0版本开始，Hadoop采用了作业级别的恢复机制^[2]。该机制不再关注各个任务的运行状态，而是以作业为单位进行恢复，它只关注两种作业状态：运行完成或者未运行完成。当JobTracker重启后，凡是未运行完成的作业将自动被重新提交到Hadoop中重新运行。除了这两种方案，学术界还尝试着研究记录级别的恢复机制^[3]。该机制尝试着从失败作业的第一条尚未处理的记录（断点）开始恢复一个任务，以尽可能地减少任务重新计算的代价。

[1] <https://issues.apache.org/jira/browse/HADOOP-3245>

[2] <https://issues.apache.org/jira/browse/MAPREDUCE-873>

[3] Jorge-Arnulfo Quian-Ruiz, Christoph Pinkel, Jörg Schäd, Jens Dittrich. RAFTing MapReduce: Fast recovery on the RAFT. In Serge Abiteboul, Klemens Böhm, Christoph Koch, Kian-Lee Tan, editors, Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany.

6.5.2 TaskTracker容错

TaskTracker负责执行来自JobTracker的各种命令，并将命令执行结果定时汇报给它。在一个Hadoop集群中，TaskTracker数量通常非常多，设计合理的TaskTracker容错机制对于及时发现存在问题的节点显得非常重要。Hadoop提供了三种TaskTracker容错机制，分别是超时机制、灰名单与黑名单机制和Exclude list与Include list。

1.超时机制

超时机制是一种在分布式环境下常用的发现服务故障的方法。如果一种服务在一定时间未响应，则可认为该服务出现了故障，从而启动相应的故障解决方案。Hadoop也采用了类似的方法发现出现故障的TaskTracker，具体如下：

- TaskTracker第一次汇报心跳后，JobTracker会将其放入过期队列trackerExpiryQueue中，并将其加入网络拓扑结构中。

- TaskTracker以后每次汇报心跳，JobTracker均会记录最近心跳时间（TaskTrackerStatus.lastSeen）

- 线程expireTrackersThread周期性地扫描过期队列trackerExpiryQueue，如果发现某个TaskTracker在10分钟（可通过参数mapred.tasktracker.expiry.interval配置）内未汇报心跳，则将其从集群中移除。

移除TaskTracker之前，JobTracker会将该TaskTracker上所有满足以下两个条件的任务杀掉，并将它们重新加入任务等待队列中，以便被调度到其他健康节点上重新运行。

条件1 任务所属作业处于运行或者等待状态。

条件2 未运行完成的Task（包括Map Task和Reduce Task）或者Reduce Task数目不为零的作业中已运行完成的Map Task。

注意 所有运行完成的Reduce Task和无Reduce Task的作业中已运行完成的Map Task无须重新运行，因为它们将结果直接写入HDFS中；而包含Reduce Task的作业中已运行完成的Map Task仍需重新运行，因为正常的TaskTracker无法通过HTTP获取死亡TaskTracker上的本地磁盘数据，具体原理可参考第7章。

2.灰名单与黑名单机制

这两种名单中的TaskTracker均不可以再接收作业，也就是，被宣判死亡（尽管可能还活着，但由于短时间内性能表现“太差”，JobTracker不得不让它“休息”一会）。

通过启发式算法推断出存在问题的TaskTracker被加入灰名单，一段时间之后，这些TaskTracker将重新获得一次接收任务的机会。

通过用户设定的脚本监控发现存在问题的TaskTracker被加入黑名单，这些TaskTracker不会再“复活”，直到监控脚本发现TaskTracker又活过来了。

（1）灰名单

每个作业在运行过程中会动态生成TaskTracker黑名单（一个TaskTracker列表），而位于黑名单中的TaskTracker将不会再有运行该作业的任何任务的机会。TaskTracker黑名单生成的方法是，作业在运行过程中记录每个TaskTracker使其失败的Task Attempt数目，一旦该数目超过mapred.max.tracker.failures（默认是4），对应的TaskTracker会被加入该作业的黑名单中。

JobTracker将记录每个TaskTracker被作业加入黑名单的次数#blacklist。当某个TaskTracker同时满足以下条件时，将被加入JobTracker的灰名单中：

条件1 #blacklist大小超过mapred.max.tracker.blacklists值（默认为4）。

条件2 该TaskTracker的#blacklists大小超过所有TaskTracker的#blacklist平均值的mapred.cluster.average.blacklist.threshold（默认是50%）倍。

条件3 当前灰名单中TaskTracker的数目小于所有TaskTracker数目的50%。

JobTracker为每个潜在存在问题的TaskTracker（#blacklist大于0）维护了一个环形桶数据结构。该数据结构保存了最近一段时间内TaskTracker对应的#blacklist值，由于该值随着时间推移不断变化，因此TaskTracker可能会不断进出灰名单。

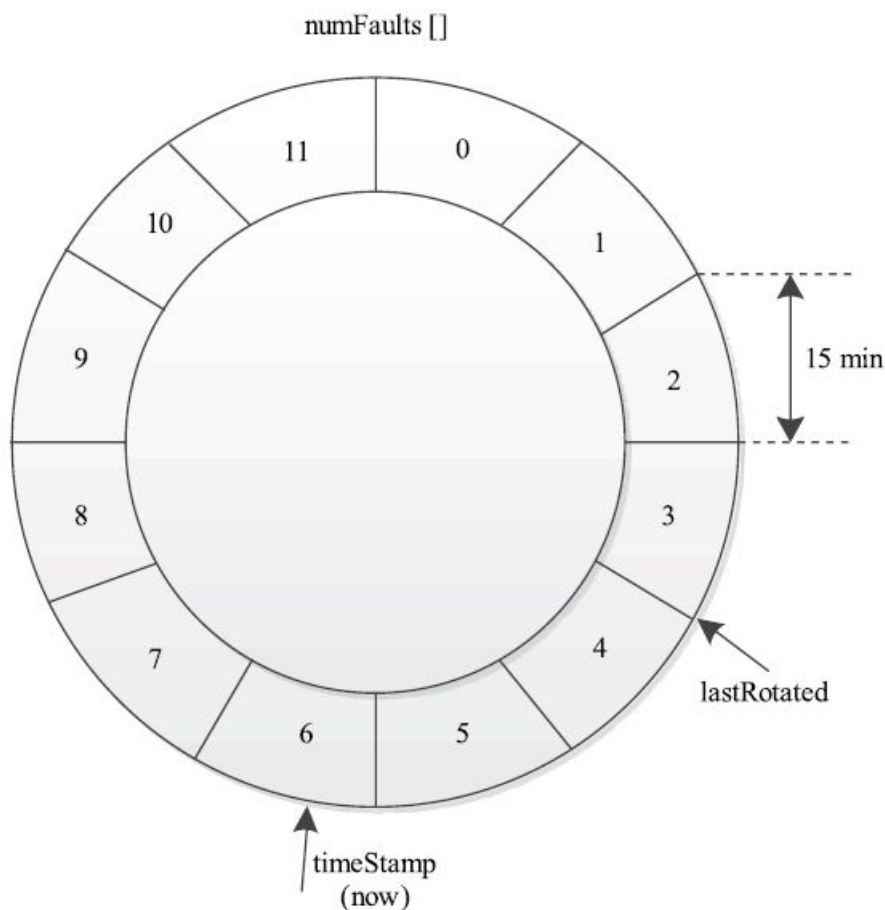


图 6-6 环形桶数据结构

一个典型的环行桶数据结构（具体参考类JobTracker.FaultInfo）如图6-6所示。默认情况下，它维护了最近mapred.jobtracker.blacklist.fault-timeout-window（默认是3小时）时间内某个TaskTracker对应的#blacklist值。为了便于计算，环形桶被分成若干个等时间片（由参数mapred.jobtracker.blacklist.fault-bucket-width配置，默认是15分钟）长度的桶，所有桶的#blacklist值由整型数组numFaults[]维护，同时由指针lastRotated指向最近一次更新所在桶的第1个毫秒位置，具体操作如下：

□初始化操作：

```
lastRotated= (time/bucketWidth) *bucketWidth; /*其中，time为当前时间，
bucketWidth为桶宽度，经初始化后，lastRotated是bucketWidth的整数倍*/
```

□checkRotation操作：将lastRotated到某个新时间点timeStamp之间的桶计数器（#blacklist）清零，同时将lastRotated移动到新时间点对应的桶第一毫秒所在位置。

```
void checkRotation (long timeStamp) {
    long diff=timeStamp-lastRotated;
    while (diff>bucketWidth) {
        //lastRotated指向时间最久的桶（它即将成为最新的桶）第一个毫秒的位置
    }
}
```

```
lastRotated+=bucketWidth;
//取得桶下标
int idx= (int) ((lastRotated/bucketWidth) %numFaultBuckets);
//清空桶计数器，为写入新值做准备
numFaults[idx]=0;
diff-=bucketWidth;
}
}
```

❑ **incrFaultCount**操作：将某个时间点对应的桶计数器加1，对应代码如下。

```
void incrFaultCount (long timeStamp) {
checkRotation (timeStamp); //将lastRotated~timeStamp时间段内桶计数器清零
++numFaults[bucketIndex (timeStamp)];
}
int bucketIndex (long timeStamp) {
return (int) ((timeStamp/bucketWidth) %numFaultBuckets);
}
```

(2) 黑名单

Hadoop允许用户编写一个脚本（**health check script**）^[1]监控TaskTracker是否健康（TaskTracker可能仍然活着，但是不健康，比如资源耗光、关键服务挂掉等），并由TaskTracker通过心跳将该脚本的检测结果汇报给JobTracker，一旦发现不健康，JobTracker会将该TaskTracker加入黑名单中，此后不再向其分配任务，直到脚本检测结果为健康。具体实现见7.3节。

3.Exclude list与Include list

Exclude list是一个非法节点列表，所有位于该列表中的节点将无法与JobTracker连接（在RPC层抛出异常）。**Include list**是一个合法节点列表（类似于节点白名单），只有位于该列表中的节点才允许向JobTracker发起连接请求。默认情况下，这两个列表是空的，表示允许任何节点接入JobTracker。这两个名单中的节点均由管理员配置，并可以动态加载生效。

管理员可在配置文件mapred-site.xml中配置Exclude list和Include list，一个简单的实例如下：

```
<property>
<name>mapred.hosts</name>
<value>/etc/hadoop_hosts/include_hosts</value>
<description>合法节点所在文件，如果文件为空或者未配置，则表示所有节点均合法。</description></property>
<property>
<name>mapred.hosts.exclude</name>
<value>/etc/hadoop_hosts/exclude_hosts</value>
<description>非法节点所在文件，如果文件为空或者未配置，则表示所有节点均合法。</description></property>
```

其中，**include_hosts**和**exclude_hosts**两个文件均保存了一个节点host列表，实例如下：

```
node0000
node0001
node0002
```

注意 黑名单与非法节点列表是两个不同的概念，区别主要有两个。

❑ **范围不同**：黑名单是TaskTracker级别的，而非法节点列表是host（一个host上可以有多个TaskTracker）级别的。

❑ **任务运行结果不同**：如果一个TaskTracker被动态添加到黑名单中，则它上面正在运行的任务可以正常运行结束（但不会为之分配新任务），但被加入非法节点列表的节点则不同，它上面所有正在运行的任务将无法成功运行完成。

综上所述，影响一个Hadoop集群中TaskTracker数量的因素如图6-7所示，管理员可根据需要，将一些节点动态加入集群或者移出集群，以更好地维护Hadoop集群或者提升它的计算能力。

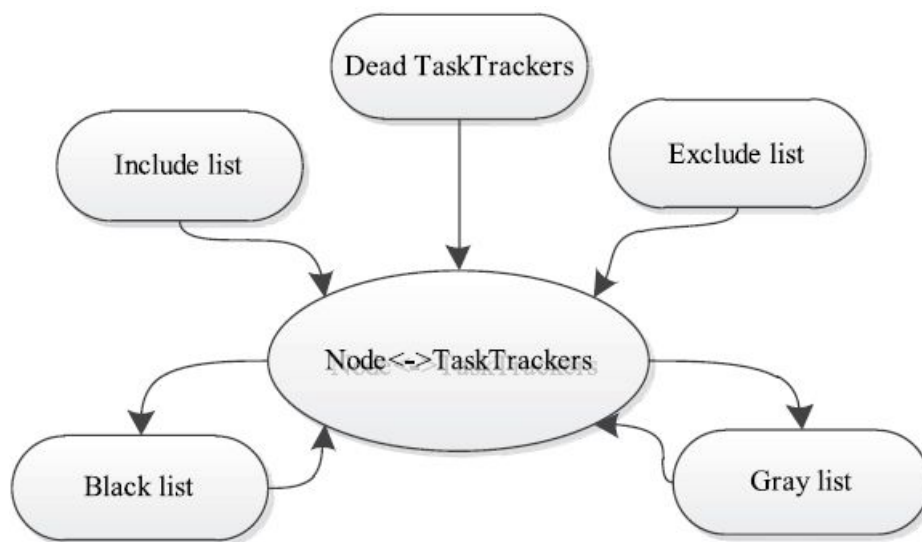


图 6-7 Hadoop集群中TaskTracker数量变化影响因素

[1] <https://issues.apache.org/jira/browse/MAPREDUCE-211>

6.5.3 Job/Task容错

1.Job容错机制

在MapReduce框架中，一个作业会被分解成多个任务，当所有任务成功运行完成时，作业才算运行成功。但在很多实际应用场景中，比如搜索引擎日志分析、网页处理等，由于数据量巨大，丢弃一小部分数据可能并不会影响最终结果。正因为如此，为支持容错，MapReduce允许丢弃部分输入数据而保证绝大部分数据有效，也就是说，MapReduce可允许部分任务失败，而其对应的处理结果不再计入最终的结果。

Hadoop为作业提供了两个可配置参数：`mapred.map.failures.percent`和`mapred.reduce.failures.percent`。用户提交作业时可通过这两个参数设定允许失败的Map任务和Reduce任务数所占总任务数的百分比。默认情况下，这两个参数值均为0，即只要有一个Map任务或者Reduce任务失败，整个作业便运行失败。

2.Task容错机制

前面提到，每个Task运行状况由对应的一个TaskInProgress对象跟踪，它允许一个Task尝试运行多次，每次称为一个运行尝试，即Task Attempt。Hadoop提供了两个可配置参数：`mapred.map.max.attempts`和`mapred.reduce.max.attempts`。用户提交作业时可通过这两个参数设定Map Task和Reduce Task尝试运行最大次数。默认情况下，这两个值均为4，即每个Task可最多运行4次，如果尝试4次之后仍旧未运行成功，则TaskInProgress对象将该任务运行状态标注为FAILED。

未运行成功的Task Attempt可分为两种：`killed task`和`failed task`，它们分别对应状态KILLED和FAILED。其中，`killed task`是MapReduce框架主动杀死的Task Attempt，一般产生于以下3种场景。

❑ 人为杀死Task Attempt：用户使用命令“`bin/hadoop job-kill-task<task-id>`”将一个Task Attempt杀死。

❑ 磁盘空间或者内存不够：任务运行过程中，出现磁盘或者内存磁盘不足，MapReduce框架需采用一定策略杀死若干个Task以释放资源。其选择杀死Task的策略是优先选择Reduce Task，其次是进度最慢的Task。

❑ TaskTracker丢失：如果TaskTracker在一定时间内未向JobTracker汇报心跳，则JobTracker认为该TaskTracker已经死掉，它上面的所有任务将被杀掉。

Failed task是自身运行失败的Task Attempt，通常产生于以下几种场景。

❑ 人为使Task Attempt失败：用户使用命令“`bin/hadoop job-fail-task<task-id>`”使一个Task Attempt杀死。

❑ 本地文件读写错误：Task运行过程中，由于磁盘坏道等原因，导致文件读写错误。

❑ Shuffle阶段错误：Reduce Task从Map Task端远程读取数据过程中出错。

❑ Counter数目过多：用户自定义的Counter或者Counter Group数目超过系统要求的上限。

❑ 一定时间内未汇报进度：由于程序Bug或者数据格式问题，任务在一定时间间隔（可通过参数`mapred.task.timeout`配置，默认10分钟）内未汇报进度。

❑ 使用内存量超过期望值：用户提交作业时指定每个Task预期使用的内存量，如果在运行过程中超过该值，则会运行失败。

❑ 任务运行过程中的其他错误：如初始化错误，其他可能的致命错误。

Failed task和killed task除了产生场景不同以外，还有以下两个重要区别。

❑调度策略：一个Task Attempt在某个节点上运行失败之后，调度器便不会再将同一个Task的Task Attempt分配给该节点；而一个Task Attempt被杀掉后，仍可能被调度到同一个节点上运行。

❑尝试次数：前面提到的Task容错机制是针对failed Task的，也就是说，任何一个Task允许的失败次数是有限的；而对于killed Task，由于它们是被框架主动杀死的，它们自身并不存在问题，因此会不断尝试运行，直到运行成功。

6.5.4 Record容错

MapReduce采用了迭代式处理模型。它将输入数据解析成一个个key/value进行迭代处理，然而，在数据处理过程中，可能由于存在一些坏记录，导致任务总是运行失败。为此，MapReduce引入了Record级别的容错机制。它能够“有记忆”地运行任务，即它会记录前几次任务尝试中导致任务失败的Record，并在下次运行时自动跳过这些坏记录 [1]。

在实际应用场景中，有多种原因使得坏记录导致任务运行失败，常见原因有两个：

- 某些记录的key或者value超大，导致内存溢出（Out Of Memory, OOM）。
- 用户应用程序使用了第三方的jar包或者静态库/动态库（不可获取源代码），由于这些程序中存在Bug，使得某些记录总是处理失败，进而导致任务运行崩溃或者任务悬挂 [2]（一旦任务长时间无响应，TaskTracker会将其杀掉）。

对于第一种情况，MapReduce允许用户在InputFormat组件中设置key或者value的最大长度（如果使用TextInputFormat，可配置参数mapred.linerecordreader.maxlength），一旦超过该长度，则直接截断字符串以防止OOM。

对于第二种情况，MapReduce采用了一种智能的有记忆尝试运行机制。前面提到，每个任务会尝试运行多次，直到任务运行成功或者达到运行次数上限。对于一个任务，MapReduce会先尝试运行几次，如果总是失败，则会自动进入skip mode模式。在该模式下，每个Task Attempt不断将接下来要处理的数据区间发送给TaskTracker，再由TaskTracker通过心跳发送给JobTracker，因此，JobTracker时刻保存了尚未处理完成的数据所在区间，这样，如果因某条坏记录导致任务运行失败，JobTracker很容易推断出坏记录所在区间。当重新运行失败任务时，JobTracker将过去识别出的所有坏记录区间“告诉”新的Task Attempt，从而可在运行过程中自动跳过这些坏记录区间。通过这种机制，Hadoop以丢失少量坏记录为代价保证整个任务运行成功，这对于很多数据密集型作业（比如日志分析）是可以接受的。

用户可通过SkipBadRecords类控制该机制。它提供了表6-3所示的几个可配置参数。

表 6-3 跳过坏记录功能控制参数

参数名称	参数含义	默认值
mapred.skip.attempts.to.start.skipping	当任务失败次数达到该值时，才会进入 skip mode，即启用跳过坏记录功能	2
mapred.skip.map.max.skip.records	用户可通过该参数设置最多允许跳过的记录数目	0（不启用跳过坏记录功能）
mapred.skip.reduce.max.skip.groups	用户可通过该参数设置 Reduce Task 最多允许跳过的记录数目	0（不启用跳过坏记录功能）
mapred.skip.out.dir	检测出的坏记录存放目录（一般为 HDFS 路径）。Hadoop 将坏记录保存起来以便于用户调试和追踪	\${mapred.output.dir}/_logs/

设mapred.skip.attempts.to.start.skipping值为k（k>=0），mapred.skip.map.max.skip.records值为N（N>0），mapred.map.max.attempts值为M（M>k），failedRanges为坏记录区间列表，保存了已运行失败的Task Attempt检测出的坏记录区间。以Map Task为例，跳过坏记录工作流程可分为以下几个步骤：

步骤1 每个任务先尝试运行k次，如果任务运行成功则停止，否则进入skip mode，令i←k+1并进入步骤2。

步骤2 第i个Task Attempt不断地（通常是每处理一条汇报一次）将接下来要处理的数据区间Range[offset, length] [3]汇报给TaskTracker，TaskTracker将之保存到变量nextRecordRange中。需要注意的是，Task Attempt会判断接下来要处理的数据是否在坏记

录区间列表failedRanges中，如果是，则跳过对应区间。

步骤3 TaskTracker通过心跳将每个任务最近的nextRecordRange值汇报给JobTracker。

步骤4 如果第i个Task Attempt运行失败，则JobTracker将查看最近一次数据处理区间长度是否超过N，如果是，则将其不断二等分，直到区间长度小于N，并依次选择这几个区间作为新Task Attempt的输入数据，以期望这些Task Attempt探测出失败记录所在的区间。设其中第j个Task Attempt运行失败，则它所处理的数据区间即为坏记录所在区间，JobTracker将该区间添加到坏记录区间列表failedRanges中。

步骤5 令 $i \leftarrow (i+j)$ ，并将最新的failedRanges值作为下一个Task Attempt的已知信息，重复步骤2~4，直到 $i \geq M$ 或者任务运行成功。

下面介绍Task Attempt如何锁定将要处理的数据区间。对于每个处于skip mode的Task Attempt而言，均包含两个指针用于锁定接下来要处理的数据区间：currentRecStartIndex和nextRecIndex。它们分别表示下一条将被RecordReader解析的数据记录索引（前面的数据已确认被成功处理完）和已被RecordReader解析但未交给Mapper/Reducer处理的记录索引。区间Range[currentRecStartIndex, nextRecIndex-currentRecStartIndex+1]为将要处理的数据区间。其中，nextRecIndex值的增加由Hadoop框架控制，而currentRecStartIndex通常由用户控制，它的值随着Hadoop内部已定义好的两个计数器值的改变而改变，这两个计数器分别是位于SkippingTaskCounters组的MapProcessedRecords和ReducerProcessedRecords中。这两个计数器在不同类型的应用程序中控制方法不同。以Map Task为例，对于Java应用程序而言，如果采用默认的MapRunner，则每处理完一条记录后，会自动对MapProcessedRecords计数器加1；然而对于Pipes/Streaming应用程序而言，由于数据处理逻辑通常由另外一种语言（非Java语言）实现，用户可能在Mapper中对记录进行缓存，因而需要用户在应用程序中根据实际逻辑增加该计数器值。

为了帮助读者更深入了解跳过坏记录的工作原理，我们接下来举一个简单的例子。假设用户需要处理一个文本文件skip-bad-records-test.txt，它的每一行是一个字符串，如果某一行内容是“Bad”，则认为它是坏记录，否则是正常记录，直接输出即可。文件内容举例如下：

```
Good
Good
Good
.....
Bad
.....
```

为了方便，我们使用Awk语言编写mapper.awk脚本作为Mapper:

```
#!/bin/awk-f
{
if ($1~/^bad$/){#这一行是坏记录
exit 1; #模拟异常退出
}else{
print"reporter: counter: SkippingTaskCounters, MapProcessedRecords, 1"\
>"/dev/stderr"; #通过标准错误输出修改Counter
print$1; #输出结果
}
}
```

我们使用Hadoop Streaming运行以上程序，Shell运行脚本如下：

```
HADOOP_HOME=/opt/dongxicheng/hadoop-1.0.0
$HADOOP_HOME/bin/hadoop jar\
$HADOOP_HOME/contrib/streaming/hadoop-streaming-1.0.0.jar\
-D mapred.job.name="Skip-Bad-Records-Test"\
-D mapred.map.tasks=1\
-D mapred.reduce.tasks=0\
-D mapred.skip.map.max.skip.records=1\
-D mapred.skip.attempts.to.start.skipping=2\
-D mapred.map.max.attempts=6\
-input"/test/input/skip-bad-records-test.txt"\
-output"/test/output"\
-mapper"mapper.awk"
```

假设输入文件中只在361行（从第0行开始计算）中有一条坏记录。根据跳过坏记录算法，仅有的一个Map Task需要尝试运行5次才会最终运行成功，如图6-8所示，过程如下：

- 1) 前两个Task Attempt尝试处理该文件，但每次到361行均异常退出，导致任务运行失败。
- 2) 从第三个Task Attempt开始进入skip mode。该Task Attempt在处理数据过程中，会不断将接下来的数据处理区间汇报给TaskTracker，再由TaskTracker汇报给JobTracker，当处理到第361行时出现错误，此时，JobTracker最后收到的数据处理区间是Range[361, 2]^[4]。
- 3) 由于数据处理区间长度超过1（一次最多可跳过坏记录条数为1），JobTracker采用二分法将该区间分裂成两段，分别是Range[361, 1]和Range[362, 1]，并将第四个Task Attempt作为测试任务，指定其数据处理区间为Range[361, 1]，即跳过区间Range[0, 361]和Range[362, ∞]，只处理第361行记录。
- 4) 第四个Task Attempt仍然运行失败，此时JobTracker可推断出Range[361, 1]为坏记录所在区间，同时将Range[362, 1]标注为正常数据区间，并将该信息传递给第五个Task Attempt。
- 5) 第五个Task Attempt在运行过程中跳过坏记录区间Range[361, 1]，最终运行成功。

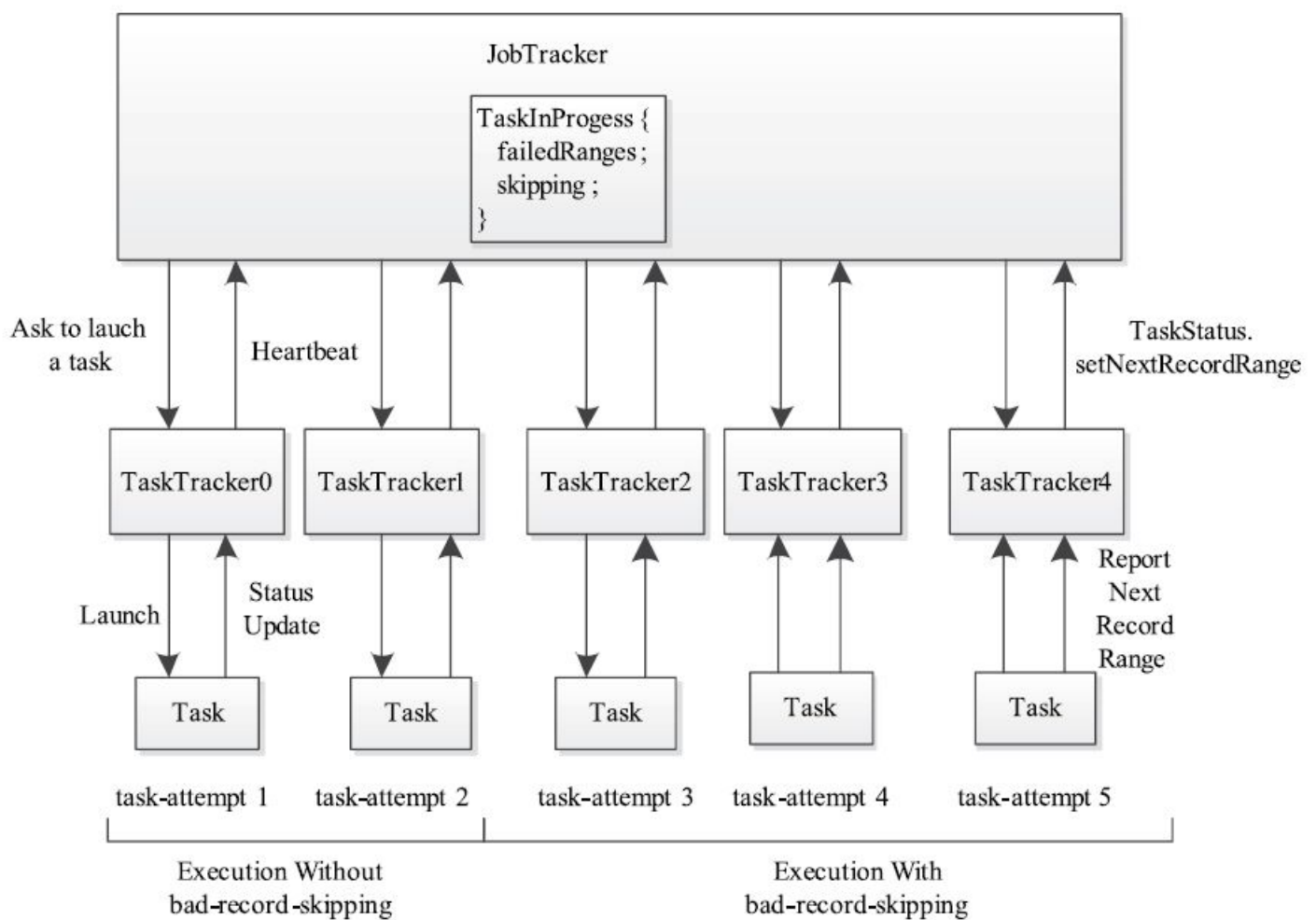


图 6-8 一个跳过坏记录实例

[1] <https://issues.apache.org/jira/browse/HADOOP-153>

[2] 任务悬挂是一种常见的现象，通常对外表现为任务阻塞，不再汇报进度和状态。

[3] `Range[offset, length]`: 表示一个数据处理区间，其中`offset`为区间中第一条记录在整个数据块中的偏移量（以记录为单

位)，`length`为区间长度。

[4] 由于存在操作系统缓存，Awk脚本程序向Hadoop框架传递计数器时不能一个一个传递，即数据区间长度大于1。

6.5.5 磁盘容错

在MapReduce中，任务需要频繁往磁盘上写数据，比如Map Task需将数据写到本地磁盘，Reduce Task需将数据写到最终的HDFS上。由于磁盘故障率明显高于其他硬件（比如内存、CPU等），因而设计合理的磁盘容错机制对于成功运行一个数据密集型作业尤为重要。

MapReduce中存在多个可配置选项用于设定各种数据输出目录，这些选项大多同时支持配置多个目录，为了提高写效率和负载均衡，用户通常将不同磁盘挂载到这些目录，一个典型的架构如图6-9所示。

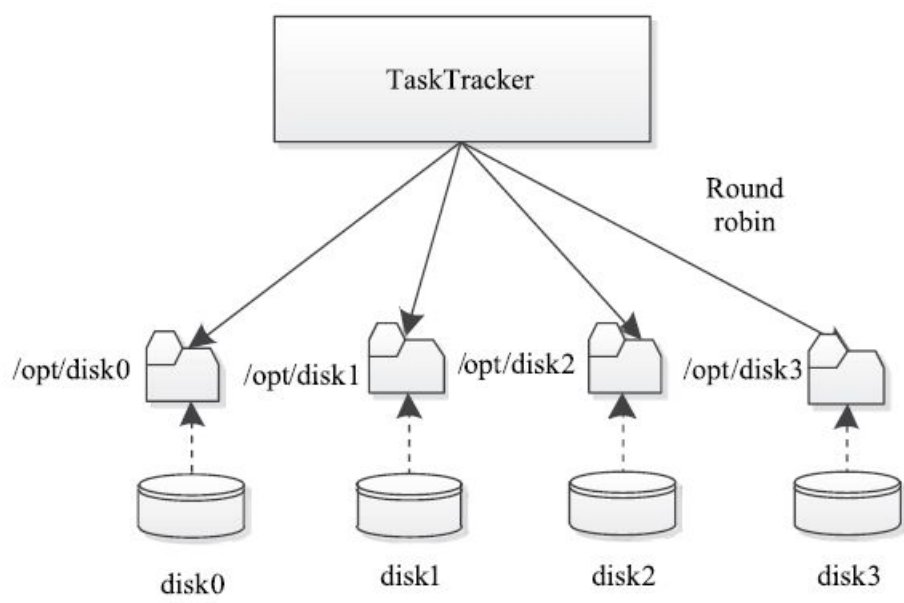


图 6-9 磁盘容错架构

TaskTracker有多种可选策略可将数据均衡地写到这些磁盘，比如轮询选择、随机选择、最多剩余空间磁盘优先等。当前TaskTracker实现中采用了轮询策略，即轮流选择磁盘作为任务的输出目录所在位置。该策略与随机选择和最多剩余空间磁盘优先策略比较，可明显降低产生写热点（大量任务同时往一个磁盘上写数据）的可能性，有利于实现写负载均衡。

(1) TaskTracker采用的机制

TaskTracker允许用户设定以下两个参数以保证节点上有足够的可用磁盘空间，防止任务因磁盘空间不足而运行失败。

❑ `mapred.local.dir.minspacestart`: TaskTracker需要保证的最小可用磁盘空间。只有当可用磁盘空间超过该参数值时，才会接收新任务。

❑ `mapred.local.dir.minspacekill`: TaskTracker会周期性检查所在节点的剩余磁盘空间，一旦低于该阈值，便会按照一定策略杀掉正在运行的任务以释放磁盘空间。

对于MapReduce而言，最重要的目录是用于保存Map Task中间结果的目录，它由参数`mapred.local.dirs`指定。TaskTracker刚启动时，首先会检查这些目录的健康状况，并将健康目录保存下来以便使用，这之后，它还会周期性（周期由参数`mapred.disk.healthChecker.interval`指定，默认是60秒）检查各个目录的健康状况，一旦发现某个正常目录出现故障（比如属性变为只读），则会重新对自己进行初始化（该过程与JobTracker向TaskTracker发送`ReinitTrackerAction`命令后，TaskTracker重启过程一致，涉及清理磁盘空间、初始化各种服务等）。

(2) JobTracker采用的机制

JobTracker上保存了所有任务的运行时信息。它可以通过已经运行完成的任务产生的数据量估算出其他同作业任务需要的磁盘空间，这可以防止因为某个节点磁盘空间不足以容纳某个任务运行结果而造成任务运行失败。

JobTracker采用的数据量估算方法如下。

当一个作业已经运行完成的Map Task数目超过Map Task总数的1/10时，JobTracker开始估算剩余Map Task产生的数据量，它采用了以下简单的线性模型：

$$\text{estimatedTotalMapOutput} = \text{inputSize} \times \frac{\text{completedMapsOutputSize}}{\text{completedMapsInputSize}} \times 2$$
$$\text{estimatedMapOutput} = \text{estimatedTotalMapOutput} / \text{numMapTasks}$$

其中，inputSize表示输入数据总量，completedMapsInputSize/completedMapsOutputSize表示所有已经运行完成的Map Task的总输入数据量和总输出数据量，最后乘2表示输出数据量会翻倍（保守估计）。

在此基础上，可估算出Reduce Task的输入数据量：

$$\text{estimatedReduceInput} = \text{estimatedTotalMapOutput} / \text{numReduceTasks}$$

如果估算到某个Map/Reduce Task产生的数据量或者输入的数据量超过某个TaskTracker剩余磁盘空间，则不会将该Task分配给它。

6.6 任务推测执行原理

在分布式集群环境下，因为程序Bug、负载不均衡或者资源分布不均等原因，会造成同一个作业的多个任务之间运行速度不一致。有些任务的运行速度可能明显慢于其他任务（比如一个作业的某个任务进度只有50%，而其他所有任务已经运行完毕），则这些任务会拖慢作业的整体执行进度。为了避免这种情况发生，Hadoop采用了推测执行（Speculative Execution）机制。它根据一定的法则推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该任务与原始任务同时处理同一份数据，并最终选用最先成功运行完成任务的计算结果作为最终结果。

在MapReduce应用程序中，用户可分别通过参数`mapred.map.tasks.speculative.execution`和`mapred.reduce.tasks.speculative.execution`控制是否对Map Task和Reduce Task启用推测执行功能。默认情况下，这两个参数均为`true`，表示启用该功能。

6.6.1 计算模型假设

Hadoop在设计之初隐含了一些假设，而正是这些假设影响了Hadoop最初的推测执行设计算法。总结起来，共有以下5个假设：

□ 每个节点的计算能力是一样的。

□ 任务的执行进度随时间线性增加。

□ 启动一个备份任务的代价可以忽略不计。

□ 一个任务的进度可以表示成已完成工作量占总工作量的比例（位于0~1之间）。对于Map Task而言，可表示成已读数据量占总数据量（任务对应的数据分片大小）的比例；对于Reduce Task而言，可将其分为三个子阶段：Shuffle、Sort和Reduce，每个阶段各占总时间的1/3。在每个阶段内部，其进度的计算方法跟Map Task一样，总结如下：

$$\text{progress} = \begin{cases} M/N, & \text{For Map Task} \\ 1/3 \times (K + M/N), & \text{For Reduce Task} \end{cases}$$

其中，M表示已读取的数据量；N表示总数据量；K=0, 1, 2，分别对应Shuffle、Sort和Reduce三个阶段。比如，一个Reduce Task位于Reduce阶段，且已读取数据量为120 MB，总数量为200 MB，则进度为 $1/3 \times (2 + 120/200) = 86.7\%$ 。

□ 同一个作业同种类型的任务工作量是一样的，所用总时间相同。

很明显，这些假设完全是基于同构集群和负载均衡的前提下，一旦集群异构或者负载不均衡（比如不同Reduce Task任务之间计算量差距很大），则很多机制将会产生问题。

6.6.2 1.0.0版本的算法

如果一个任务满足以下条件，则会为该任务启动一个备份任务：

□ 该任务尚未进入skip mode（由于推测执行机制和跳过坏记录机制均会减慢任务执行进度，考虑到性能问题，不会同时启用这两个功能）。

□ 该任务没有其他正在运行的备份任务（当前Hadoop最多允许一个任务同时启动两个Task Attempt）。

□ 该任务已经运行时间超过60秒且当前正在运行的Task Attempt落后（同一个作业所有Task Attempt的）平均进度的20%，即对于任意一个任务i，如果满足：

$$\text{progress}[i] < \text{progress}_{\text{avg}} - 20\%$$

其中， $\text{progress}_{\text{avg}} = \sum_{i=1}^T \text{progress}[i] / T$

则任务i将被当作“拖后腿”任务，进而需为其启动备份任务。

当该任务的某个Task Attempt成功运行完成后，JobTracker会杀掉另外一个Task Attempt。

该版本实现的推测执行功能存在很多问题，以下是几个常见问题。

□ 适用情况考虑不全：当作业的大部分任务已经运行完成时，如果存在若干个Task Attempt的运行进度等于或者超过80%，则此时总不会启动备份任务。

□ 缺乏保证备份任务执行速度的机制：由于新启动的备份任务需要首先处理原始Task Attempt已经处理完的数据，因此需保证备份任务的运行速度不低于原始Task Attempt，否则将失去启动备份任务的意义。

□ 参数不可配置：比如上面的数值“60秒”和“20%”均不可配置，这不能满足用户根据自己集群特点定制参数的要求。

6.6.3 0.21.0版本的算法

为了解决1.0.0版本中存在的问题，Hadoop 0.21.0提出了新的算法LATE（Longest Approximate Time to End）^[1]。对于“适用情况考虑不全”问题，它采用了基于任务运行速度和任务最大剩余时间的策略，尽可能地提高发现“拖后腿”任务的可能性；对于“缺乏保证备份任务执行速度的机制”问题，它根据历史任务运行速度对节点进行性能评测，以识别出快节点和慢节点，并将新启动的备份任务分配给快节点；对于“参数不可配置”问题，它增加了多个配置选项，使一些常量数据尽可能地可配置，进而方便用户按照自己的应用特点和集群特点定制相应的参数值。

1.配置选项

Hadoop 0.21.0中增加了三个配置选项，用户在提交某个作业时可根据需要指定这三个参数值。

(1) `mapreduce.job.speculative.slownodethreshold`

这个参数用于定义该作业在任意一个TaskTracker上已运行完成任务的平均进度增长率（一个任务在单位时间内运行进度的增量）与所有已运行完成任务的平均进度增长率的最大允许差距（标准方差的倍数）。如果超过该阈值，则认为对于该作业而言，该TaskTracker性能过低，不会在其上启动一个备份任务。该配置选项默认值是1，表示如果一个TaskTracker上已运行完成任务的平均进度增长率与所有已运行完成任务的平均进度增长率的差距超过所有已运行完成的任务进度增长率标准方差的1倍，则不会在该TaskTracker上为该作业启动任何备份任务。

(2) `mapreduce.job.speculative.slowtaskthreshold`

这个参数用于定义该作业的任意一个任务平均进度增长率与所有正在运行任务的平均进度增长率的最大允许差距（标准方差的倍数）。当超过该阈值时，则认为该任务运行过慢，需为之启动一个备份任务。其默认值是1，表示如果一个任务平均进度增长率与（同一个作业的）所有任务平均进度增长率的差距超过所有任务进度增长率标准方差的1倍，则需为该任务启动一个备份任务。

(3) `mapreduce.job.speculative.speculativecap`

这个参数用于限定该作业允许启动备份任务的任务数目占正在运行任务的百分比。其默认值为0.1，表示可为一个作业启动推测执行功能的任务数不能超过正在运行任务的10%。

2.启动备份任务

下面介绍对于一个作业J与TaskTracker X，如何判断是否能够在X上为J的某个任务（比如任务T）启动一个备份任务。首先引入以下几个变量：

□ $\text{progressRate}(O)_s$ ：作业J中所有运行状态为s（可以为f或者r，分别表示已经运行完成的任务和正在运行的任务）的任务的进度增长率，O为任务集合。如果O为某个TaskTracker，则表示该TaskTracker上所有运行状态为s的任务平均进度增长率；如果O为*，则表示整个集群中所有运行状态为s的任务平均进度增长率。对于任意一个任务T，它的任务进度增长率计算方法为：

$$\text{progressRate}_T = \frac{\text{progress}_T}{\text{currentTime} - \text{dispatchTime}_T}$$

其中 progress_T 是当前任务执行进度， currentTime 为系统当前时间， dispatchTime_T 任务T被调度的时间。

□ σ_s ：作业J所有任务进度增长率的标准方差。

□ `slowNodeThreshold/slowTaskThreshold/speculativeCap`: 分别对应以上三个配置选项的参数值。

何时选择并启动备份任务是由任务选择策略决定的，具体可参考6.7.2小节。在JobTracker端，`JobInProgress`类中的`findSpeculativeTask`方法用于选择一个需启动备份任务的Task。它通过以下四步发现一个“拖后腿”任务：

步骤1 判断X是否是一个慢TaskTracker，如果是，则不能启动任何备份任务。

为了判断一个TaskTracker是否适合启动备份任务，Hadoop通过该TaskTracker运行作业J的其他任务时的性能表现对其进行评估，如果满足以下条件，则认为该TaskTracker有能力启动一个备份任务：

$$\overline{\text{progressRate}(X)}_f - \overline{\text{progressRate}(*)}_f \leq \sigma_f \times \text{slowNodeThreshold}$$

步骤2 检查作业J已经启动的任务数是否超过限制。

由于一个任务一旦启动了备份任务，则需要两倍的计算资源处理同样的数据，为了防止推测执行机制滥用，Hadoop要求同时启动的备份任务数目与所有正在运行任务的比例不能超过`speculativeCap`，即满足以下条件：

$$\text{speculativeTaskCount}/\text{numRunningTask} < \text{speculativeCap}$$

其中，`speculativeTaskCount`表示作业J已经启动的备份任务数目，`numRunningTask`表示作业J正在运行的任务总数。

步骤3 筛选出作业J中满足以下条件的所有任务，并保存到数组`candidates`中。

□ 该任务未在TaskTracker X上运行失败过。

□ 该任务没有其他正在运行的备份任务。

□ 该任务已运行时间超过60秒。

□ 该任务已经出现“拖后腿”的迹象，主要判断准则是：

$$\overline{\text{progressRate}(*)}_f - \overline{\text{progressRate}}_f \leq \sigma_f \times \text{slowNodeThreshold}$$

步骤4 按照运行剩余时间由大到小对`candidates`中的任务进行排序，并选择剩余时间最大任务为其启动备份任务。

当数组`candidates`中有多个待选任务时，Hadoop倾向于选择剩余时间最长的任务，因为这样的任务使得其备份任务替代自己的可能性最大。为此，Hadoop采用了一个简单的线性模型估算一个任务的剩余时间`timeLeft`：

$$\text{timeLeft} = \frac{\text{progressRate}}{1 - \text{progress}}$$

LATE算法并不是完美的。在实际使用时，由于LATE算法采用了静态方式计算任务的进度（对应前面的假设3），可能导致性能仍然比较低，主要体现在以下两个方面。

□ 任务进度和任务剩余时间估算不准确：这会导致部分正常任务被误认为是“拖后腿”任务，从而造成资源浪费。

□ 未针对任务类型对节点分类：尽管LATE算法可通过任务执行速度识别出慢节点，但它未分别针对Map Task和Reduce Task做出更细粒度的识别。而实际应用中，一些节点对于Map Task而言是慢节点，但对Reduce Task而言则是快节点。

为了解决这些问题，论文《SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment》^[2]在LATE算法

基础上进行了改进，提出了SAMR（Self Adaptive MapReduce）算法。该算法通过历史信息调整各个参数以提高估算任务进度和任务剩余时间的准确性，同时分别针对Map Task和Reduce Task将节点分成快节点和慢节点。

- [1] Matei Zaharia, Andrew Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica, Improving MapReduce Performance in Heterogeneous Environments, 8th USENIX Symposium on Operating Systems Design Implementation, pp.29-42, San Diego, CA, December, 2008.
- [2] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, Song Guo, "SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment, "Computer and Information Technology (CIT) , 2010 IEEE 10th International Conference.

6.6.4 2.0版本的算法

Apache MapReduce 2.0（简称MRv2，也称为YARN，即Yet Another Resource Negotiator）属于下一代MapReduce计算框架，我们将在第12章进行详细介绍。MapReduce 2.0采用了不同于以上两种算法的推测执行机制，它重点关注新启动的备份任务是否有潜力比当前正在运行的任务完成得更早。如果通过一定的算法推测某一时刻启动备份任务，该备份任务肯定会比当前任务完成得晚，那么启动该备份任务只会浪费更多的资源。然而，从另一个角度看，如果推测备份任务比当前任务完成得早，则启动备份任务会加快数据处理，且备份任务完成得越早，启动备份任务的价值越大。

假设某一时刻，任务T的执行进度为progress，则可通过一定的算法推测出该任务的最终完成时刻estimatedEndTime。另一方面，如果此刻为该任务启动一个备份任务，则可推断出它可能的完成的时刻estimatedEndTime'，于是可得出以下几个公式：

$$\text{estimatedEndTime} = \text{estimatedRunTime} + \text{taskStartTime}$$

$$\text{estimatedRunTime} = (\text{currentTimestamp} - \text{taskStartTime}) / \text{progress}$$

$$\text{estimatedEndTime}' = \text{currentTimestamp} + \text{averageRunTime}$$

其中，currentTimestamp为当前时刻；taskStartTime为该任务启动时刻；averageRunTime为已经成功运行完成的任务的平均运行时间。这样，MRv2总是选择（estimatedEndTimeestimatedEndTime'）差值最大的任务，并为之启动备份任务。为了防止大量任务同时启动备份任务造成资源浪费，MRv2为每个作业设置了同时启动的备份任务数目上限。

推测执行机制实际上采用了经典的算法优化方法：以空间换时间，它同时启动多个相同任务处理相同的数据，并让这些任务竞争以缩短数据处理时间。显然，这种方法需要占用更多的计算资源。在集群资源紧缺的情况下，应合理使用该机制，争取在多用少量资源的情况下，减少大作业的计算时间。

6.7 Hadoop资源管理

Hadoop资源管理由两部分组成：资源表示模型和资源分配模型。其中，资源表示模型用于描述资源的组织方式，Hadoop采用“槽位”（slot）组织各节点上的资源；而资源分配模型则决定如何将资源分配给各个作业/任务，在Hadoop中，这一部分由一个插拔式的调度器完成。

Hadoop引入了“slot”概念表示各个节点上的计算资源。为了简化资源管理，Hadoop将各个节点上的资源（CPU、内存和磁盘等）等量切分成若干份，每一份用一个slot表示，同时规定一个Task可根据实际需要占用多个slot^[1]。通过引入“slot”这一概念，Hadoop将多维度资源抽象简化成一种资源（slot），从而大大简化了资源管理问题。

更进一步说，slot相当于任务运行“许可证”。一个任务只有得到该“许可证”后，才能够获得运行的机会，这也意味着，每个节点上的slot数目决定了该节点上的最大允许的任务并发度。为了区分Map Task和Reduce Task所用资源量的差异，slot又被分为Map slot和Reduce slot两种，它们分别只能被Map Task和Reduce Task使用。Hadoop集群管理员可根据各个节点硬件配置和应用特点为它们分配不同的Map slot数（由参数`mapred.tasktracker.map.tasks.maximum`指定）和Reduce slot数（由参数`mapred.tasktracker.reduce.tasks.maximum`指定）。

在分布式计算领域中，资源分配问题实际上是一个任务调度问题。它的主要任务是根据当前集群中各个节点上的资源（包括CPU、内存和网络等资源）剩余情况与各个用户作业的服务质量（Quality of Service）要求，在资源和作业/任务之间做出最优的匹配。由于用户对作业服务质量的要求是多样化的，因此，分布式系统中的任务调度是一个多目标优化问题，更进一步说，它是一个典型的NP问题。

在Hadoop中，由于Map Task和Reduce Task运行时使用了不同种类的资源（不同种类的slot），且这两种资源之间不能混用，因此任务调度器分别对Map Task和Reduce Task单独进行调度。而对于同一个作业而言，Reduce Task和Map Task之间存在数据依赖关系，默认情况下，当Map Task完成数目达到总数的5%（可通过参数`mapred.reduce.slowstart.completed.maps`配置）后，才开始启动Reduce Task（Reduce Task开始被调度）。

一个作业从提交到开始执行的过程如图6-10所示，整个过程大约需7步。

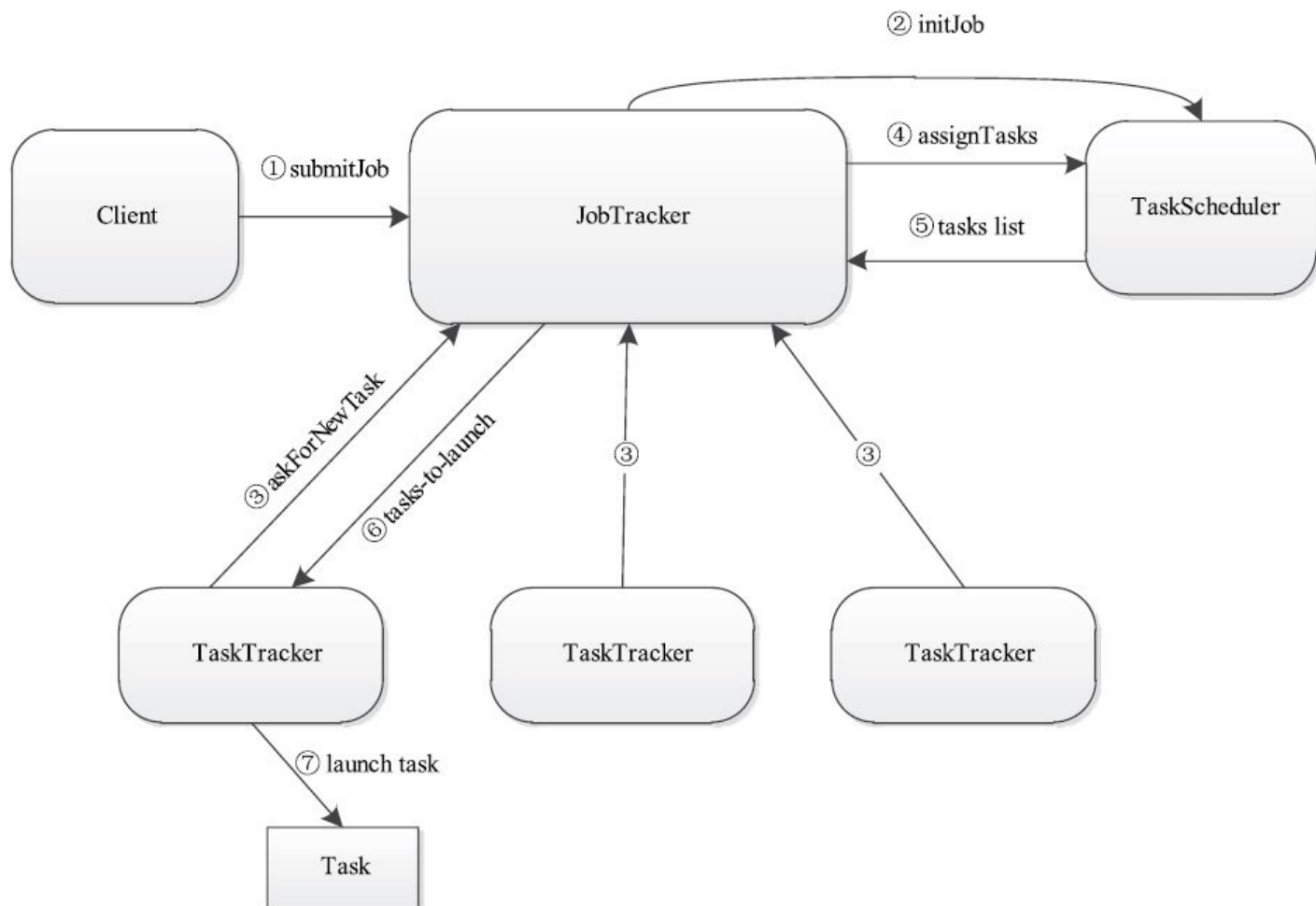


图 6-10 作业从提交到开始执行的过程

步骤1 客户端调用作业提交函数将程序提交到JobTracker端；

步骤2 JobTracker收到新作业后，通知任务调度器（TaskScheduler）对作业进行初始化；

步骤3 某个TaskTracker向JobTracker汇报心跳，其中包含剩余的slot数目和能否接收新任务等信息；

步骤4 如果该TaskTracker能够接收新任务，则JobTracker调用TaskScheduler对外函数assignTasks为该TaskTracker分配新任务；

步骤5 TaskScheduler按照一定的调度策略为该TaskTracker选择最合适的任务列表，并将该列表返回给JobTracker；

步骤6 JobTracker将任务列表以心跳应答的形式返回给对应的TaskTracker；

步骤7 TaskTracker收到心跳应答后，发现有需要启动的新任务，则直接启动该任务。

本小节重点关注Hadoop的资源分配模型，它由一个插拔式的调度器TaskScheduler实现。下一小节主要介绍TaskScheduler的架构与设计思路。

6.7.1 任务调度框架分析

在Hadoop中，任务调度是一个可插拔的模块。用户可以根据自己的实际应用需求设计调度器，然后在配置文件中指定相应的调度器（可通过参数`mapred.jobtracker.taskScheduler`配置），这样JobTracker启动时便会加载该调度器。

Hadoop提供了一个调度器公共基础类TaskScheduler，用户只需继承该类并根据需要重新实现其中的若干个函数便可以实现自己的调度器。TaskScheduler类的主要定义如下：

```

abstract class TaskScheduler implements Configurable{
.....
protected TaskTrackerManager taskTrackerManager; //实际就是JobTracker
public synchronized void setTaskTrackerManager (
TaskTrackerManager taskTrackerManager) {
this.taskTrackerManager=taskTrackerManager;
}
//初始化函数
public void start()throws IOException{
//do nothing
}
//结束函数
public void terminate()throws IOException{
//do nothing
}
//为TaskTracker分配新任务
public abstract List<Task>assignTasks (TaskTracker taskTracker)
throws IOException;
.....
}

```

在Hadoop中，任务调度器和JobTracker之间存在函数相互调用的关系，它们彼此都拥有对方需要的信息或者功能。对于JobTracker而言，它需要调用任务调度器中的assignTasks函数为TaskTracker分配新的任务，同时，JobTracker内部保存了整个集群中的节点、作业和任务的运行时状态信息，这信息是任务调度器进行调度决策时需要用到的。JobTracker与调度器之间的函数调用关系如图6-11所示，需要注意以下几点：

1) 任务调度器需要通过一个或者多个JobInProgressListener对象从JobTracker端监听作业状态的变化，包括作业添加、作业更新和作业删除等。

2) 任务调度器包括两个主要功能：作业初始化和任务调度。其中，作业初始化发生在JobInProgressListener#jobAdded (JobInProgress) 之后，TaskScheduler#assignTasks (TaskTracker) 之前，通过调用函数JobInProgress.initJob (JobInProgress) 完成。

3) 任务调度器中最重要的对外函数是assignTasks。JobTracker收到能够接收新任务的TaskTracker后，会调用该函数为它分配新任务。它的输入参数是一个TaskTracker对象，输出参数是为该TaskTracker分配的任务列表。

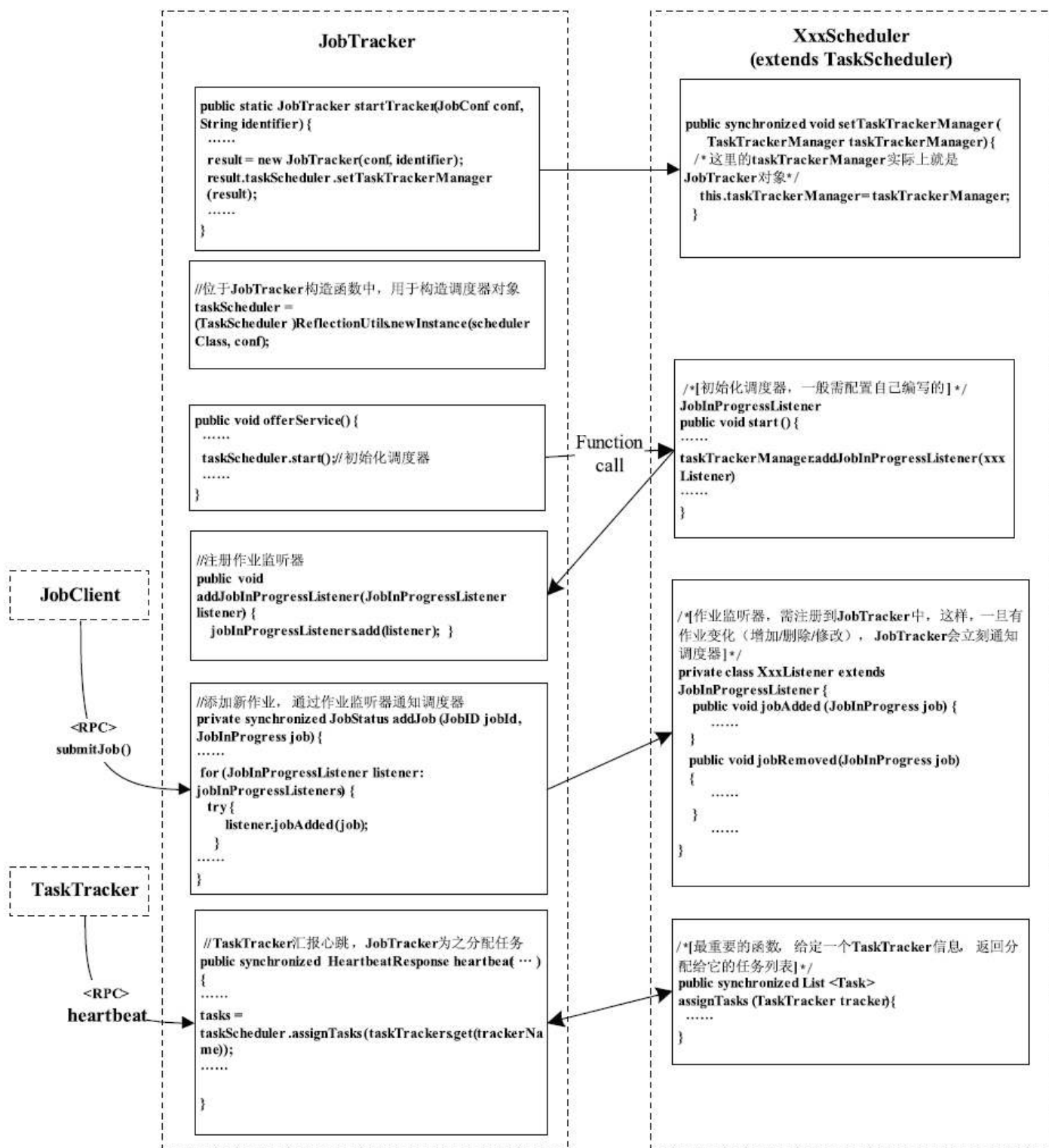


图 6-11 JobTracker与调度器之间的函数调用关系

Hadoop以队列为单位管理作业和资源，每个队列分配有一定量的资源，同时管理员可指定每个队列中资源的使用者以防止资源滥用。添加“队列”这一概念后，现有的Hadoop调度器本质上均采用了三级调度模型。如图6-12所示，当一个TaskTracker出现空闲资源时，调度器会依次选择一个队列、（选中队列中的）作业和（选中作业中的）任务，并最终将这个任务分配给TaskTracker。

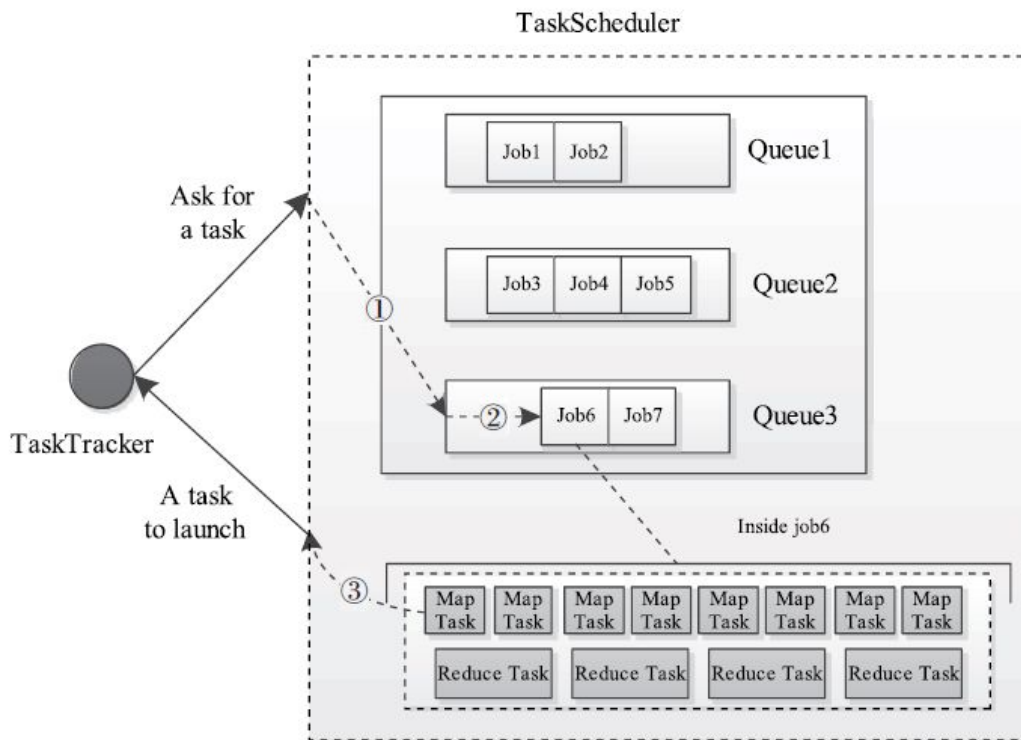


图 6-12 三级调度模型

①选择一个队列②选择一个作业③选择一个任务

在Hadoop中，不同任务调度器的主要区别在于队列选择策略和作业选择策略不同，而任务选择策略通常是相同的，也就是说，给定一个节点，从一个作业中选择一个任务需考虑的因素是一样的，均主要为数据本地性（data-locality）。总之，一个任务调度器通用的assignTasks函数伪代码实现如下：

```
//为TaskTracker分配任务，并返回任务列表
List<Task>assignTasks (TaskTracker taskTracker) :
List<Task>taskList;
while taskTracker.askForTasks() : //不断分配新的任务
Queue queue=selectAQueueFromCluster(); //从系统中选择一个队列
JobInProgress job=selectAJobFromQueue (queue); //从队列中选择一个作业
Task task=job.obtainNewTask (job); //从作业中选择一个任务
taskList.add (task);
taskTracker.addNewTask (task);
return taskList;
```

由于不同调度器采用的任务选择策略是一样的，因此Hadoop将之封装成一个通用的模块供各个调度器使用，具体存放在JobInProgress类中的obtainNewMapTask和obtainNewReduceTask方法中。我们将在下一小节分析这两个方法的实现机制。

[1] 一个Task可使用多少slot完全由调度器决定。当前大部分调度器只支持一个Task占用一个slot，比如FIFO和Fair Scheduler；而Capacity Scheduler则可根据Task内存需求为其分配多个slot。

6.7.2 任务选择策略分析

任务选择发生在调度器选定一个作业之后，目的是从该作业中选择一个最合适的任务。在Hadoop中，选择Map Task时需考虑的最重要的因素是数据本地性，也就是尽量将任务调度到数据所在节点。除了数据本地性之外，还需考虑失败任务、备份任务的调度顺序等。然而，由于Reduce Task没有数据本地性可言，因此选择Reduce Task时通常只需考虑未运行任务和备份任务的调度顺序。

(1) 数据本地性

在分布式环境中，为了减少任务执行过程中的网络传输开销，通常将任务调度到输入数据所在的计算节点，也就是让数据在本地进行计算，而Hadoop正是以“尽力而为”的策略保证数据本地性的。

为了实现数据本地性，Hadoop需要管理员提供集群的网络拓扑结构。如图6-13所示，Hadoop集群采用了三层网络拓扑结构，其中，根节点表示整个集群，第一层代表数据中心，第二层代表机架或者交换机，第三层代表实际用于计算和存储的物理节点。对于目前的Hadoop各个版本而言，默认均采用了二层网络拓扑结构，即数据中心一层暂时未被考虑。

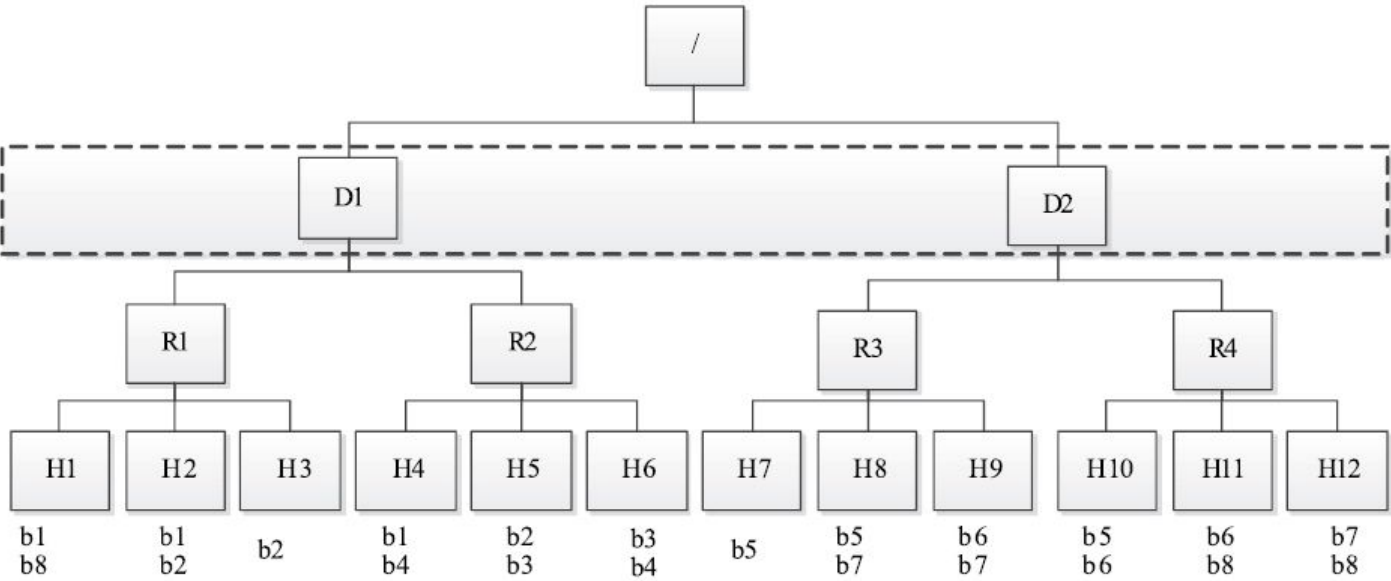


图 6-13 Hadoop网络拓扑结构图

Hadoop根据输入数据与实际分配的计算资源之间的距离将任务分成三类：**node-local**（输入数据与计算资源同节点），**rack-local**（同机架）和**off-switch**（跨机架）。当输入数据与计算资源位于不同节点上时，Hadoop需将输入数据远程复制到计算资源所在节点进行处理。两者距离越远，需要的网络开销越大，因此调度器进行任务分配时尽量选择离输入数据近的节点资源。

当Hadoop进行任务选择时，采用了自下向上查找的策略。由于当前采用了两层网络拓扑结构，因此这种选择机制决定了任务优先级从高到低依次为：**node-local**、**rack-local**和**off-switch**。下面结合图6-13介绍三种类型的任务被选中的场景。

假设某一时刻，TaskTracker X出现空闲的计算资源，向JobTracker汇报心跳请求新的任务，调度器根据一定的调度策略为之选择了任务Y。

场景1 如果X是H1，任务Y输入数据块为b1，则该任务为node-local。

场景2 如果X是H1，任务Y输入数据块为b2，则该任务为rack-local。

场景3 如果X是H1，任务Y输入数据块为b4，则该任务为off-switch。

(2) Map Task选择策略

用户追踪作业运行状态的JobInProgress对象为Map Task维护了五个数据结构：


```
//Node与未运行的TIP集合映射关系，通过作业的InputFormat可直接获取的
Map<Node, List<TaskInProgress>>nonRunningMapCache;
//Node与运行的TIP集合映射关系，一个任务获得调度机会，其TIP便会添加进来
Map<Node, Set<TaskInProgress>>runningMapCache;
//non-local且未运行的TIP集合，non-local是指任务没有输入数据（InputSplit为空），//这可能是一些计算密集型任务，比如Hadoop example中的PI作业
final List<TaskInProgress>nonLocalMaps;
//按照Task Attempt失败次数排序的TIP集合
final SortedSet<TaskInProgress>failedMaps;
//non-local且正在运行的TIP集合
Set<TaskInProgress>nonLocalRunningMaps;
```

当需要从作业中选择一个Map Task时，调度器会直接调用JobInProgress中的obtain-NewMapTask方法。该方法封装了所有调度器公用的任务选择策略实现。其主要思想是优先选择运行失败的任务，以让其快速获取重新运行的机会，其次是按照数据本地性策略选择尚未运行的任务，最后是查找正在运行的任务，尝试为“拖后腿”任务启动备份任务。具体步骤如下：

步骤1 合法性检查。如果一个作业在某个节点上失败任务数目超过一定阈值或者该节点剩余磁盘容量不足，则不再将该作业的任何任务分配给该节点。

步骤2 从failedMaps列表中选择任务。failedMaps保存了按照Task Attempt失败次数排序的TIP集合。失败次数越多的任务，被调度的机会越大。需要注意的是，为了让失败的任务快速得到重新运行的机会，在进行任务选择时不再考虑数据本地性。

步骤3 从nonRunningMapCache列表中选择任务。采用的任务选择方法完全遵循数据本地性策略，即任务选择优先级从高到低依次为node-local, rack-local和off-switch类型的任务。

步骤4 从nonLocalMaps列表中选择任务。由于nonLocalMaps中的任务没有输入数据，因此无须考虑数据本地性。

步骤5 从runningMapCache列表中选择任务。遍历runningMapCache列表，查找是否存在正运行且“拖后腿”的任务，如果有，则为其启动一个备份任务。

步骤6 从nonLocalRunningMaps列表中选择任务。同步步骤5，从nonLocalRunningMaps列表中查找“拖后腿”任务，并为其启动备份任务。

步骤2~6中任何一个步骤中查找到一个合适的任务后则直接返回，不再进行下面的步骤。

（3）Reduce Task选择策略

由于Reduce Task不存在数据本地性，因此，与Map Task相比，它的调度策略显得非常简单。JobInProgress对象为其保存了两个数据结构：nonRunningReduces和runningReduces，分别表示尚未运行的TIP列表和正在运行的TIP列表。由此采用的任务选择步骤如下：

步骤1 合法性检查。同Map Task一样，对节点可靠性和磁盘空间进行检查。

步骤2 从nonRunningReduces列表中选择任务。无须考虑数据本地性，只需依次遍历该列表中的任务，选出第一个满足条件（未曾对应节点上失败过）的任务。

步骤3 从runningReduces列表中选择任务，为“拖后腿”的任务启动备份任务。

6.7.3 FIFO调度器分析

当前Hadoop自带了多个任务调度器，最常用的是FIFO（默认调度器）、Capacity Scheduler和Fair Scheduler三种。在本小节中，我们重点介绍FIFO调度器，其他两种将在第10章介绍。

FIFO，顾名思义，即首先到达的作业优先获得调度机会。它是由类org.apache.hadoop.mapred.JobQueueTaskScheduler实现的，其类关系如图6-14所示。

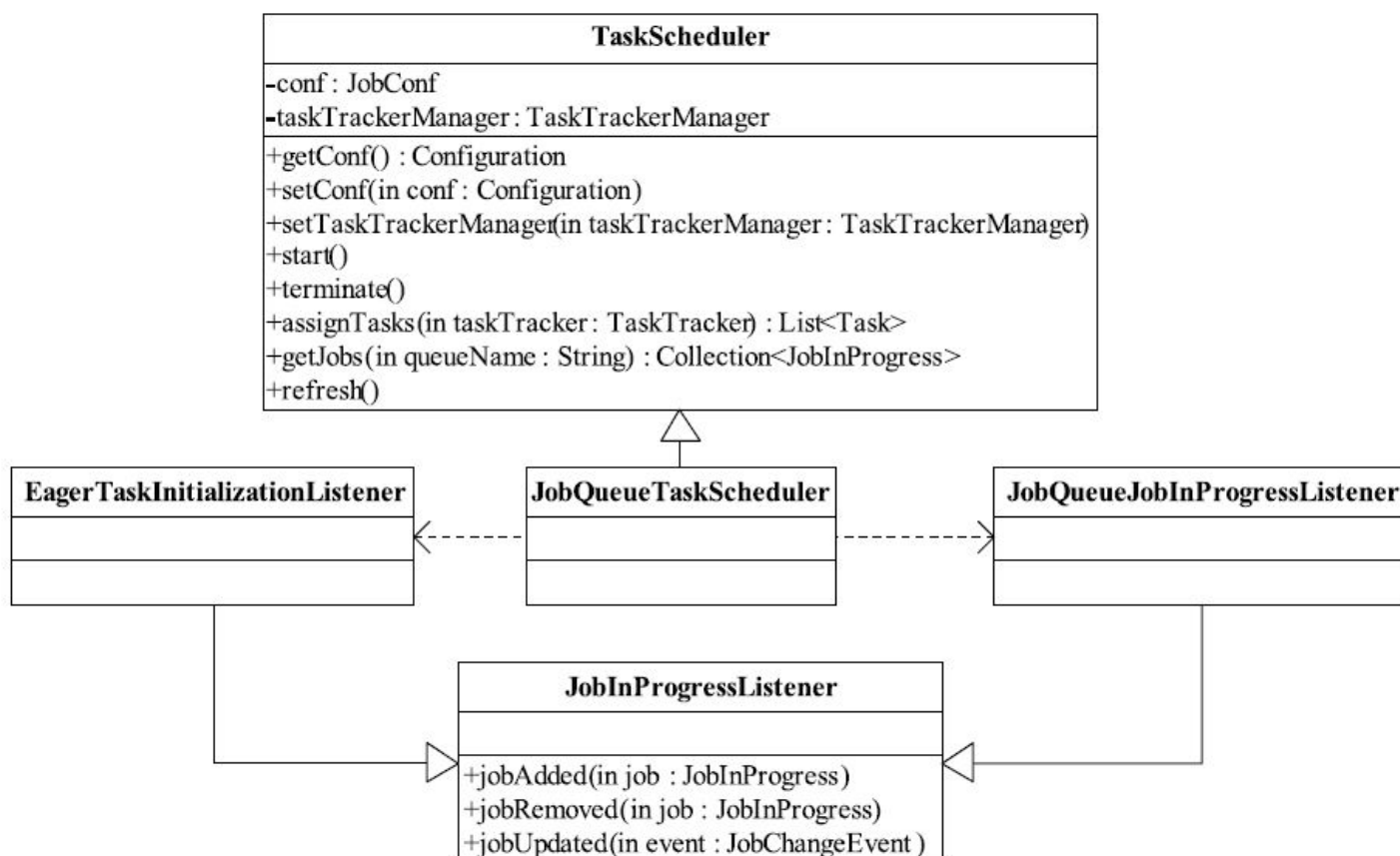


图 6-14 FIFO类关系图

JobQueueTaskScheduler同时向JobTracker注册了两种作业监听器，功能如下。

□ EagerTaskInitializationListener对用户提交的作业进行初始化：考虑到待初始化的作业可能非常多，因此需采用一定的策略决定新提交作业的初始化顺序，基本策略是优先选择优先级高的作业，当优先级相同时，优先选择提交时间早的作业。

□ JobQueueJobInProgressListener维护作业的调度顺序：该作业监听器维护了作业被调度的顺序，其排序原则跟初始化顺序类似：先按作业优先级排序，然后是作业提交时间，最后是作业ID。

基于以上两种作业监听器，JobQueueTaskScheduler的调度机制（由函数assignTasks实现）如下。

□ 计算可用slot数目：FIFO调度器尽量将所有任务均衡地调度到各个TaskTracker，以便均衡地使用各个节点上的资源。对于TaskTracker X，假设它的slot总数为trackerCapacity，当前正在运行的任务数为trackerRunningTasks，则可通过以下方法计算它当前可用的slot数目：

$$\text{loadFactor} = \min \left\{ \sum_{\text{jobs}} (\text{numTasks} - \text{finishedTasks}) / \text{totalSlots}, 1 \right\}$$

$$\text{availableSlots} = \lceil \text{loadFactor} \times \text{trackerCapacity} \rceil - \text{trackerRunningTasks}$$

其中，`numTasks`和`finishedTasks`表示作业总的任务数和已经完成的任务数，`totalSlots`表示系统的slot总数。

□分配任务：遍历作业队列（已按照调度顺序排好序），并依次调用`JobInProgress`类中的`obtainNewMapTask`和`obtainNewReduceTask`为该`TaskTracker`选择`availableSlots`个任务。

6.7.4 Hadoop资源管理优化

前面提到，Hadoop资源管理由两部分组成：资源表示模型和资源分配模型。考虑到这两部分是耦合在一起的，因此，如果想对Hadoop资源管理进行优化，则需要同时结合这两部分进行考虑。本小节介绍了三种常见的Hadoop资源管理优化方案、分别为基于动态slot的方案、基于无类别slot的方案和基于真实资源需求量的方案。

在正式介绍这几个资源优化方案之前，我们先回顾一下Hadoop 1.0中的资源管理方案。Hadoop 1.0采用了基于slot的资源表示模型。为了简化资源分配问题，Hadoop将各个节点上的多维度资源（CPU、内存、网络I/O和磁盘I/O等）抽象成一维度slot，这样便把复杂的多维度资源分配问题转换成简单的slot分配问题。此外，考虑到Map Task和Reduce Task资源使用量不同，Hadoop又进一步将slot划分成Map slot和Reduce slot两种，并规定Map Task只能使用Map slot, Reduce Task只能使用Reduce slot。具体如图6-15所示。

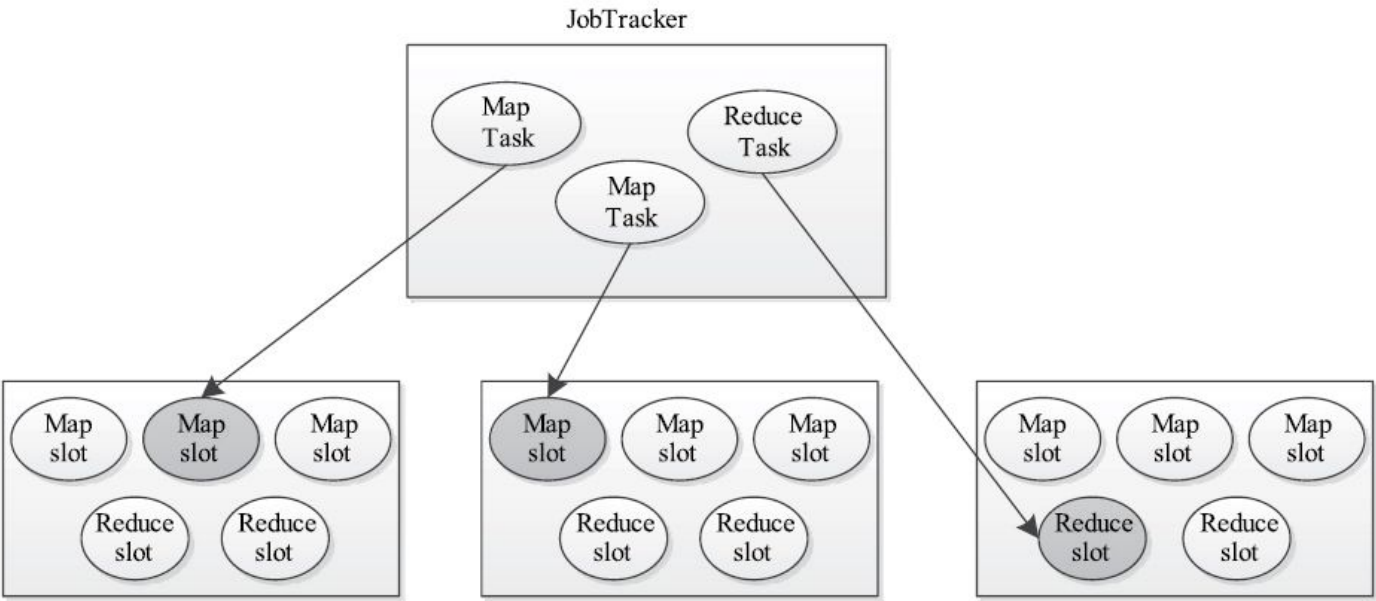


图 6-15 基于slot的资源管理方案

(1) 基于动态slot的资源管理方案

Hadoop 1.0中的资源分配方案采用了静态资源设置策略，即每个节点实现配置好可用的slot总数，这些slot数目一旦启动后无法再动态修改。考虑到实际应用场景中，不同作业对资源的需求往往具有较大差异，静态配置slot数目往往会导致节点上的资源利用率过高或者过低。为了解决该问题，可尝试采用动态调整slot数目的方案^[1]，具体如图6-16所示。该方案在每个节点上安装一个slot数目动态调整模块 SlotsAdjuster，它可以根据节点上的资源利用率动态调整slot数目，以便更合理地利用资源。

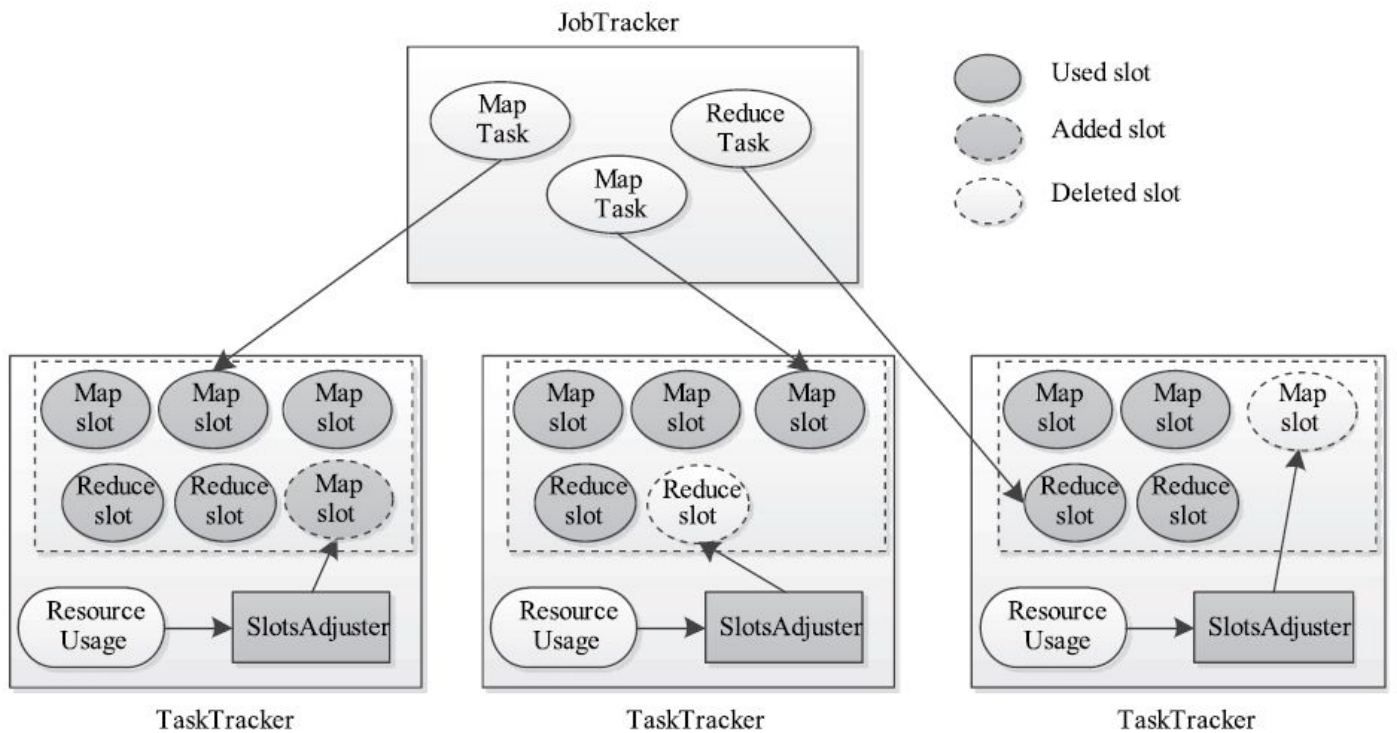


图 6-16 基于动态slot的资源管理方案

(2) 基于无类别slot的资源管理方案

对于一个MapReduce应用程序而言，Map Task优先得到调度，而只有当Map Task完成数目达到一定比例（默认是5%）后，Reduce Task才开始获得调度机会。因此，从单个应用程序看，刚开始运行时，Map slot资源紧缺而Reduce slot空闲，当Map Task全部运行完成后，Reduce slot紧缺而Map slot空闲。很明显，这种区分slot类别的资源管理方案在一定程度上降低了slot的利用率。如图6-17所示，一种直观的解决方案是不再区分Map slot和Reduce slot^[2]，而是只有一种slot，并让Map Task和Reduce Task共享这些slot；至于怎样将这些slot分配给Map Task和Reduce Task，则完全由调度器决定。

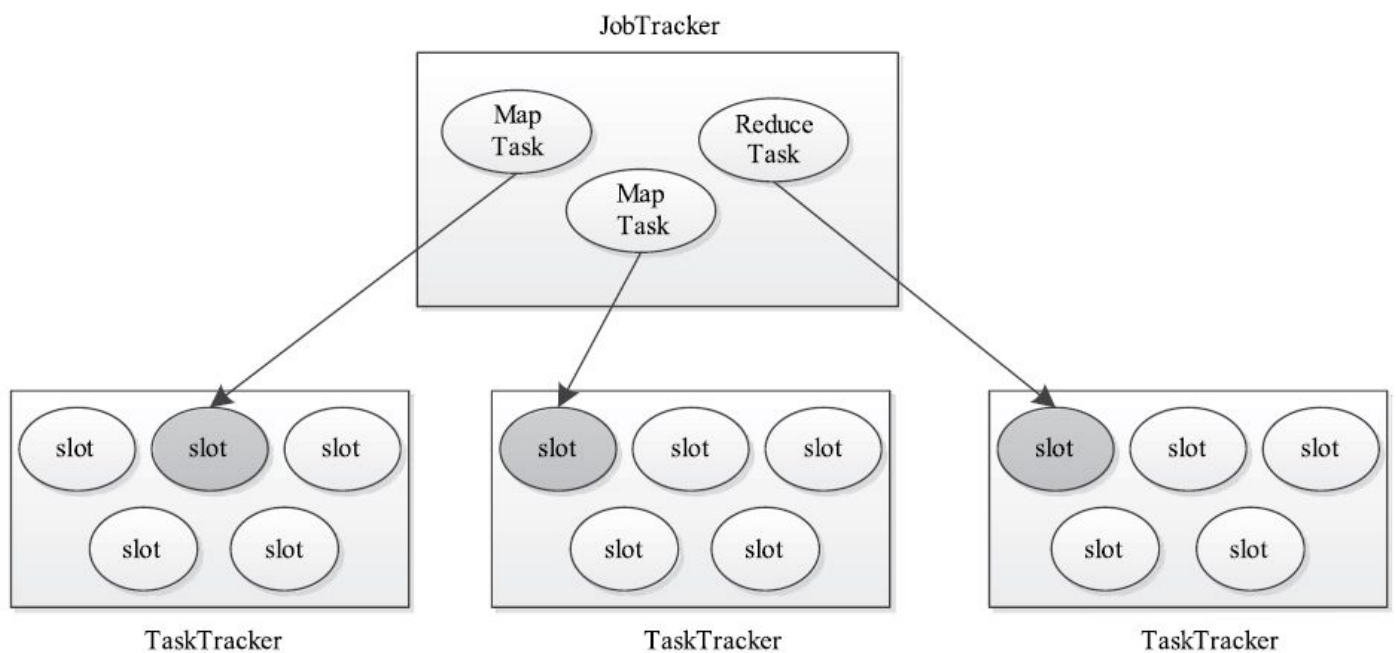


图 6-17 基于无类别slot的资源管理方案

(3) 基于真实资源需求量的资源管理方案

让我们进一步分析这种基于无类别slot的资源管理方案可能存在的问题。由于整个Hadoop集群中只有一种slot，因此，这隐含着各个TaskTracker上的slot是同质的，也就是说，一个slot实际上代表了相同的资源。然而，在实际应用环境中，用户应用程序对资源需求往往是

多样化的（不同应用程序对资源的要求不同）。这种基于无类别slot的资源划分方法的划分粒度仍过于粗糙，往往会造成节点资源利用率过高或者过低 [3]。比如，管理员事先规划好一个slot代表2 GB内存和1个CPU，如果一个应用程序的任务只需要1 GB内存，则会产生“资源碎片”，从而降低集群资源的利用率；同样，如果一个应用程序的任务需要3 GB内存，则会隐式地抢占其他任务的资源，从而产生资源抢占现象，可能导致集群利用率过高。因此，寻求一种更精细的资源划分方法显得尤为必要。

让我们回归到资源分配的本质，即根据任务资源需求为其分配系统中的各类资源。在实际系统中，资源本身是多维度的，包括CPU、内存、网络I/O和磁盘I/O等，因此。如果想精确控制资源分配，不能再有slot的概念，最直接的方法是让任务直接向调度器申请自己需要的资源（比如某个任务可申请1.5 GB内存和1个CPU），而调度器则按照任务实际需求为其精细地分配对应的资源量，不再简单地将一个Slot分配给它，具体如图6-18所示。

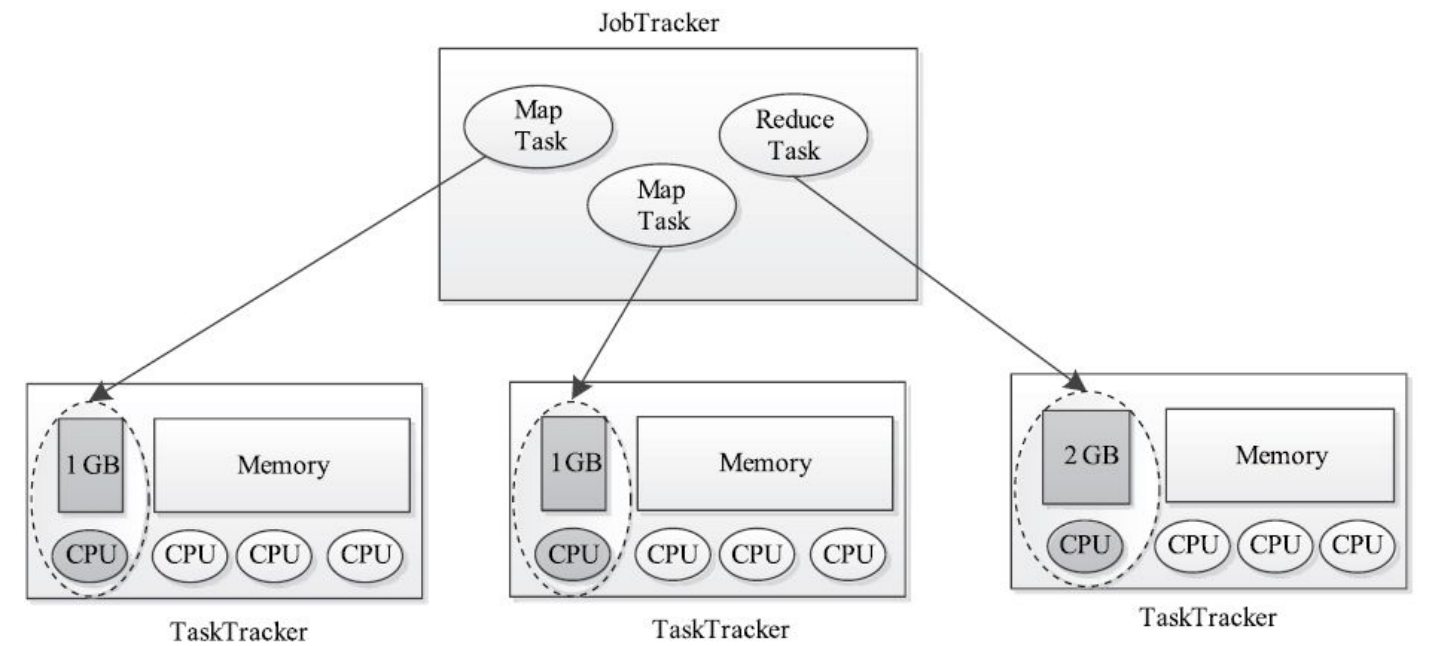


图 6-18 基于真实资源需求量的资源管理方案

Hadoop 2.0中便采用了这种完全基于真实资源需求量的资源管理方案，比如Apache YARN、Facebook Corona等（将在第12章详细介绍）。相比于基于slot的方案，这种方案更加直观，且能够大大提高资源利用率。

[1] 梁李印，《阿里Hadoop集群架构及服务体系》，PPT, Hadoop与大数据技术大会（HBTC 2012）。

[2] Hong Mao, Shengqiu Hu, Zhenzhong Zhang, Limin Xiao, Li Ruan: A Load-Driven Task Scheduler with Adaptive DSC for MapReduce. GreenCom 2011: 28-33.

[3] A. Ghodsi, M.Zaharia, B.Hindman, A.Konwinski, S.Shenker, and I.Stoica.Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.In USENIX NSDI, 2011.

6.8 小结

本章介绍了Hadoop MapReduce最核心的服务JobTracker的实现。

JobTracker是整个MapReduce计算框架中的主服务，相当于集群的“管理者”，负责整个集群的作业控制和资源管理。本章重点围绕这两个方面讲解JobTracker。

JobTracker启动过程涉及三个方面：重要对象初始化、工作线程初始化和作业恢复。启动之后，将开启RPC server以等待来自各个TaskTracker的心跳请求。

JobTracker的主要功能之一是作业控制，包括作业的分解和状态监控。其中，最重要的是状态监控，包括TaskTracker状态监控、作业状态监控和任务状态监控，其中，作业状态监控采用了“三层多叉树”模型，该模型分为三层，从高到低依次为JobInProgress、TaskInProgress和Task Attempt三种对象集合。状态监控的一个目的是容错，Hadoop设计了各个级别的容错机制，包括JobTracker容错、TaskTracker容错、Job/Task容错、Record容错和磁盘容错等。

JobTracker另外一个功能是资源管理，它由两部分组成：资源表示模型和资源分配模型。Hadoop采用了基于slot的资源表示模型，而资源分配模型实际上是任务调度模型，它由可插拔的任务调度器完成。当前大部分调度器采用了三级调度架构，即当一个TaskTracker出现空闲资源时，调度器会依次选择一个队列、（选中队列中的）作业和（选中作业中的）任务，并最终将这个任务分配给TaskTracker。本章以Hadoop默认FIFO调度器为例讲解了一个简单调度器的实现原理。

第7章 TaskTracker内部实现剖析

在Hadoop中，MapReduce采用了master/slave架构。这在第6章已提到了，并对与master对应的JobTracker进行了详细介绍。在本章中，我们将剖析slave的实现——TaskTracker。与JobTracker一样，TaskTracker也是以服务组件的形式存在的。它分布在各个slave节点上，负责任务的执行和任务状态的汇报。

本章将从TaskTracker架构、TaskTracker行为（如启动新任务，杀死任务等）、作业目录管理和任务启动等几个方面深入分析TaskTracker工作原理及其实现。

7.1 TaskTracker概述

TaskTracker是Hadoop集群中运行于各个节点上的服务。它扮演着“通信枢纽”的角色，是JobTracker与Task之间的“沟通桥梁”：一方面，它从JobTracker端接收并执行各种命令，比如运行任务、提交任务、杀死任务等；另一方面，它将本节点上的各个任务状态通过周期性心跳汇报给JobTracker，具体如图7-1所示。

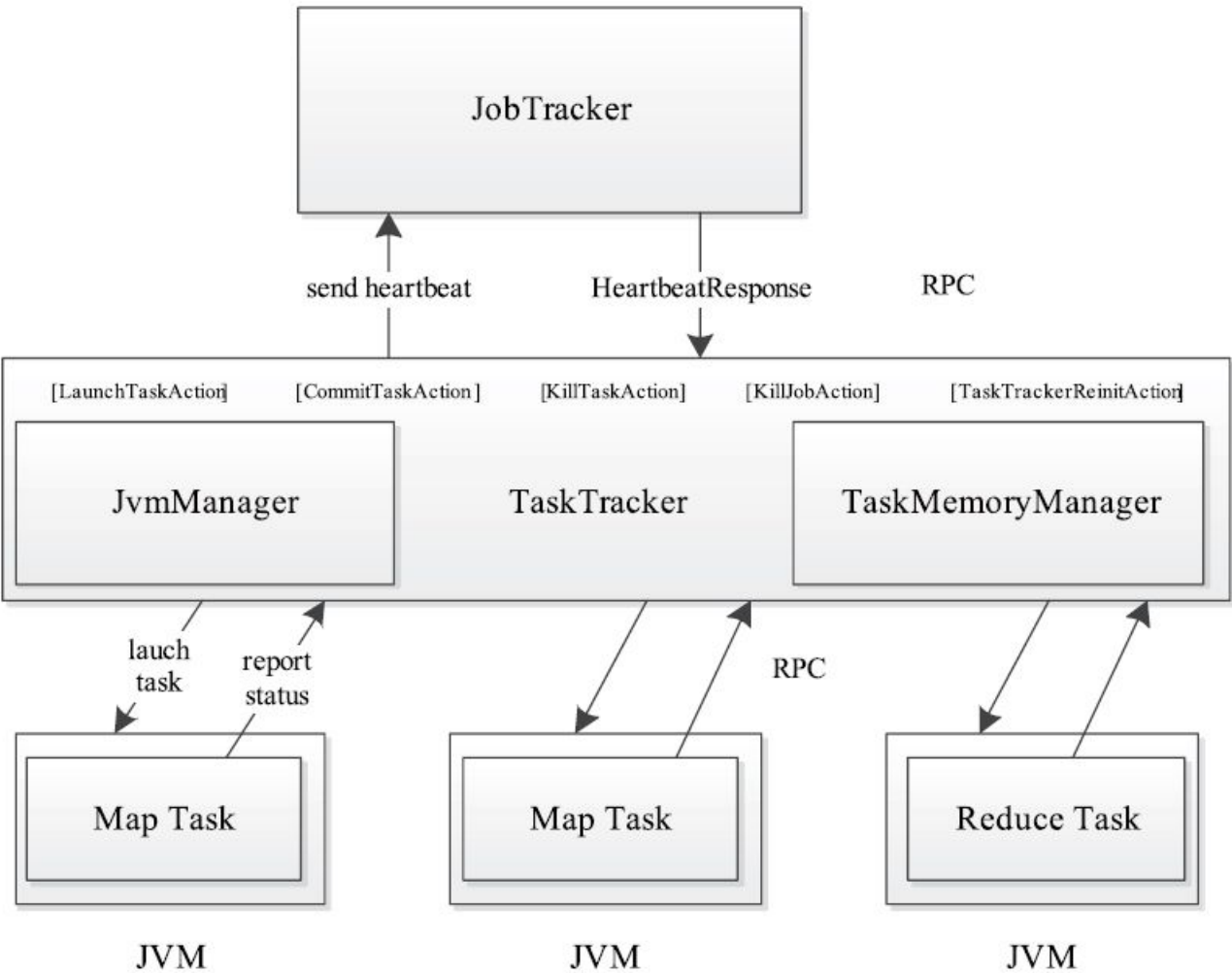


图 7-1 TaskTracker工作原理图

TaskTracker与JobTracker和Task之间采用了RPC协议进行通信。对于TaskTracker和JobTracker而言，它们之间采用InterTrackerProtocol协议，其中，JobTracker扮演RPC Server的角色，而TaskTracker扮演RPC Client的角色；对于TaskTracker与Task而言，它们之间采用TaskUmbilicalProtocol协议，其中，TaskTracker扮演RPC Server的角色，而Task扮演RPC Client的角色。这两个协议的具体内容已在第4章进行了详细介绍，本章不再重复介绍。

总体来说，TaskTracker实现了两个功能：汇报心跳和执行命令。具体如下：

（1）汇报心跳

TaskTracker周期性地将所在节点^[1]上各种信息通过心跳机制汇报给JobTracker。这些信息包括两部分：机器级别信息，如节点健康状况、资源使用情况等；任务级别信息，如任务执行进度、任务运行状态、任务Counter值等。

（2）执行命令

JobTracker收到TaskTracker心跳信息后，会根据心跳信息和当前作业运行情况为该TaskTracker下达命令，主要包括启动任务（LaunchTaskAction）、提交任务（CommitTaskAction）、杀死任务（KillTaskAction）、杀死作业（KillJobAction）和重新初始化（TaskTrackerReinitAction）5种命令。其中，过程比较复杂的是启动任务。为了防止任务之间的干扰，TaskTracker为每个任务创建一个单独的Java虚拟机（Java Virtual Machine, JVM），并有专门的线程监控其资源使用情况，一旦发现超量使用资源就直接将其杀掉。

^[1] Hadoop允许一个节点上部署多个TaskTracker，并通过端口号区别它们，但通常情况下只部署一个。

7.2 TaskTracker启动过程分析

TaskTracker服务由TaskTracker类实现。该类实现了四个接口，分别是MRConstants, TaskUmbilicalProtocol, Runnable和TaskTrackerMXBean。其中，MRConstants定义了一些常量；TaskUmbilicalProtocol定义了TaskTracker与Task之间的RPC协议；Runnable是Java库中的线程接口，实现该接口意味着将以线程形式启动TaskTracker；TaskTrackerMXBean实现了一个JMX MXBean，方便外界监控工具（比如Ganglia、Nagios等）获取TaskTracker运行时的信息。

TaskTracker是一个独立的服务，有一个对应的main函数启动它。main函数实现非常简单：创建一个TaskTracker对象并启动它。

```
public class TaskTracker implements MRConstants, TaskUmbilicalProtocol,
Runnable, TaskTrackerMXBean{
.....//成员变量定义和成员函数实现
public static void main (String argv[]) throws Exception{
.....
JobConf conf=new JobConf();
.....
TaskTracker tt=new TaskTracker (conf); //创建TaskTracker对象
MBeans.register ("TaskTracker", "TaskTrackerInfo", tt); //注册MXBean
tt.run(); //启动TaskTracker线程
.....
}
.....
}
```

TaskTracker在构造函数中初始化一些重要对象和线程，而在run方法中维护一个与JobTracker的通信连接，以周期性地向JobTracker发送心跳并领取新的任务。

接下来将展开介绍TaskTracker的启动过程，包括重要变量初始化、重要对象初始化等。

7.2.1 重要变量初始化

TaskTracker类中包含很多成员变量，用于管理节点和监控节点上的任务。TaskTracker启动时会初始化这些变量。下面介绍几个非常重要的变量的含义。

```
volatile boolean running=true; //TaskTracker是否正在运行
InterTrackerProtocol jobClient; //RPC Client, 用于与JobTracker通信
short heartbeatResponseId=-1; //心跳响应ID
TaskTrackerStatus status=null; //TaskTracker状态信息
Map<TaskAttemptID, TaskInProgress>tasks=new HashMap<TaskAttemptID,
TaskInProgress> (); //该节点上TaskAttemptID与TIP对应关系
Map<TaskAttemptID, TaskInProgress>runningTasks=null; /*该节点上正在运行的
TaskAttemptID与TIP对应关系*/
//该节点正运行的作业列表。如果一个作业中的任务在节点上运行，则把该作业加入该数据结构
Map<JobID, RunningJob>runningJobs=new TreeMap<JobID, RunningJob> ();
boolean acceptNewTasks=true; //是否接受新任务，每次汇报心跳时要将该值告诉JobTracker
private int maxMapSlots; //TaskTracker上配置的Map slot数目
private int maxReduceSlots; //TaskTracker上配置的Reduce slot数目
private volatile int heartbeatInterval=HEARTBEAT_INTERVAL_MIN; //心跳间隔
```

7.2.2 重要对象初始化

TaskTracker构造函数内部对一些重要对象进行了初始化，具体可见表7-1。

表 7-1 TaskTracker 需初始化的对象列表

对象名	对应类名	意义解释
aclsManager	ACLsManager	作业访问权限控制
server	HttpServer	将 TaskTracker 相关信息显示到 Web 前端
taskController	TaskController	用于控制任务的初始化，终结和清理工作
userLogManager	UserLogManager	用户日志管理
jvmManager	JvmManager	JVM 管理
taskReportServer	Server	RPC Server，Task 通过 RPC 向该 Server 汇报进度
distributedCacheManager	TrackerDistributedCacheManager	分布式缓存管理
jobClient	InterTrackerProtocol	RPC Client，TaskTracker 通过该 Client 向 JobTracker 汇报心跳
mapEventsFetcher	MapEventsFetcherThread	获取已运行完成的 Map Task 列表，为 Reduce Task 远程拷贝数据做准备

(续)

对象名	对应类名	意义解释
taskMemoryManager	TaskMemoryManagerThread	任务内存监控
taskLauncher	TaskLauncher	启动任务
localizer	Localizer	任务本地化，即准备任务运行环境
healthChecker	NodeHealthCheckerService	节点健康状况检查
jettyBugMonitor	JettyBugMonitor	应对 Jetty 中存在的 bug 临时启用的一个监控线程

我们将在后面几节中介绍表中这些类的具体实现。

7.2.3 连接JobTracker

前面几章中提到TaskTracker与JobTracker之间通过心跳机制通信。TaskTracker服务初次启动后，会向JobTracker发出第一个心跳信息，经过JobTracker一系列安全检查后，将被添加到Hadoop集群中，进而被分配各种任务。我们将在下一节中详细介绍心跳机制。

7.3 心跳机制

7.3.1 单次心跳发送

第6章中已经提到，JobTracker与TaskTracker之间采用了pull通信模型，即JobTracker从不会主动与TaskTracker通信，而总是被动等待TaskTracker汇报信息并领取其对应的命令。TaskTracker周期性地向JobTracker汇报信息并领取任务形成心跳。

在TaskTracker类的run方法中维护了一个无限循环，用于通过心跳发送状态信息和接收命令，代码框架如下：

```
public void run() {
    .....
    while (running && ! shuttingDown && ! denied) {
        .....
        while (running && ! staleState && ! shuttingDown && ! denied) {
            .....
            State osState = offerService();
            .....
        }
    }
}
```

其中，offerService函数代码框架如下：

```
State offerService() throws Exception {
    .....
    while (running && ! shuttingDown) {
        //判断是否到达心跳发送时间
        .....
        //发送心跳
        HeartbeatResponse heartbeatResponse = transmitHeartBeat(now);
        //执行心跳响应heartbeatResponse中的各种命令
        .....
        markUnresponsiveTasks(); //杀死一定时间内未汇报进度的任务
        killOverflowingTasks(); //剩余磁盘空间小于mapred.local.dir.minspacekill时，寻找合适的任务将其杀掉以释放空间
        .....
    }
}
```

TaskTracker的单次心跳发送过程如图7-2所示，可分为以下几个步骤。

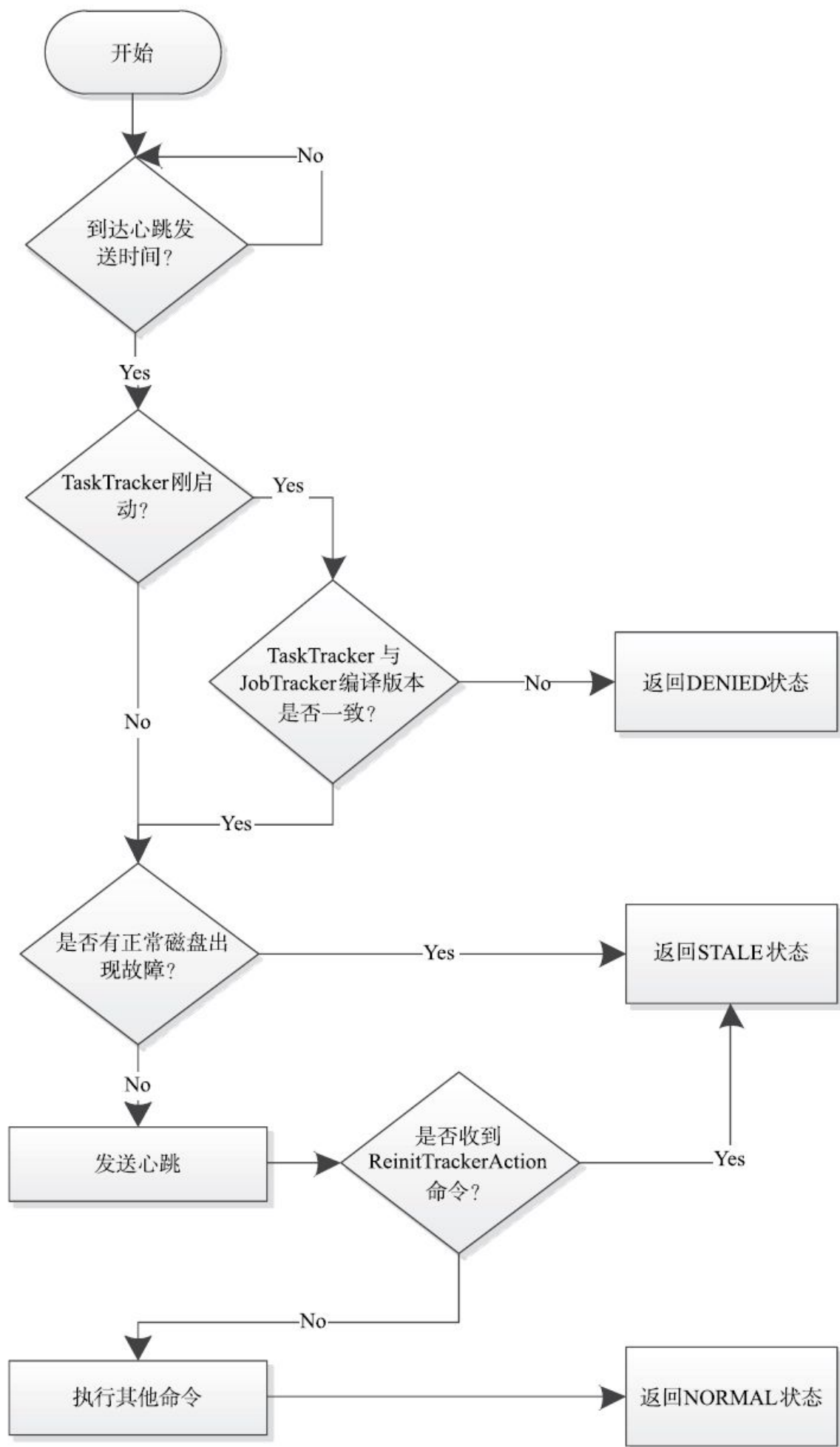


图 7-2 TaskTracker单次心跳发送过程

步骤1 判断是否到达心跳发送时间。

TaskTracker的心跳间隔由集群规模和任务运行情况共同决定。

1) 集群规模: JobTracker能够根据当前集群规模(TaskTracker数量)动态调整TaskTracker的心跳间隔,并将下一次心跳间隔放到TaskTracker的本次心跳应答中。

2) 任务运行情况: 为了提高任务的响应时间和资源利用率, TaskTracker一旦发现存在某个任务运行完成或者失败,就会立即缩短心跳间隔,以便将任务完成或失败的消息告诉JobTracker,进而快速重新分配任务,我们将这种特殊的心跳称为“带外心跳”。TaskTracker包含以下两个配置参数设置带外心跳。

❑ `mapreduce.tasktracker.outofband.heartbeat`: 决定是否启用带外心跳机制。默认值是`false`,表示不启用。

❑ `mapreduce.tasktracker.outofband.heartbeat.damper`: 心跳收缩因子。默认值是1 000 000。当启用带外心跳机制时,如果某时刻有X个任务运行完成,则心跳间隔变为 $\text{heartbeatInterval} / (X * \text{oobHeartbeatDamper} + 1)$

其中, `heartbeatInterval`是从JobTracker端获取的心跳间隔; `oobHeartbeatDamper`是心跳收缩因子(`mapreduce.tasktracker.outofband.heartbeat.damper`对应的值)。

步骤2 如果TaskTracker刚启动,则需要检查代码编译版本与JobTracker是否一致。

只有与JobTracker具有相同代码编译版本号的TaskTracker才能够向JobTracker发送心跳。代码编译版本号由Hadoop版本号、修订版本号、代码编译用户和校验和四部分组成,如“1.0.0-dev from 451451 by dongxicheng source checksum e54b3f6cb07ea1cd833d1ab0b947ac39”,其中,“1.0.0-dev”表示Hadoop版本号,“451451”表示修订版本号,“dongxicheng”表示代码编译用户,“e54b3f6cb07ea1cd833d1ab0b947ac39”表示校验和。

步骤3 检查是否有磁盘损坏。

MapReduce计算过程中最重要的输出目录是参数`mapred.local.dir`指定的中间结果存放目录(通常由多个目录组成,每个目录对应一个磁盘块)。由于这些目录存放在本地磁盘且没有备份,因此一旦损坏或者丢失后,需重新计算。TaskTracker初始化时会检查`mapred.local.dir`指定的磁盘目录列表,并将正常目录存放起来,之后,TaskTracker周期性(时间间隔由`mapred.disk.healthChecker.interval`指定,默认是60 s)检查这些正常目录,如果发现出现故障的目录,则TaskTracker会重新对自己初始化。

步骤4 发送心跳。

TaskTracker将当前节点运行时信息,比如资源使用情况、任务运行状态等,通过心跳汇报给JobTracker,同时接收来自JobTracker的各种命令。

步骤5 接收并执行命令。

JobTracker收到TaskTracker的心跳信息后,会为之下达命令。

接下来我们重点分析发送心跳和命令执行两个过程。

7.3.2 状态发送

TaskTracker通过心跳向JobTracker汇报的是当前节点运行时信息，包括TaskTracker基本信息、节点资源使用情况和各个任务状态等，这些信息被封装到可序列化类TaskTrackerStatus中。每次发送心跳时，TaskTracker会根据最新信息重新构造一个TaskTrackerStatus，且每次包含的信息量可能不一样。比如，任务的计数器信息每隔60 s才会发送一次，且只有当askForNewTask为true时，才会发送节点资源使用信息。其中，askForNewTask值的计算方法如下：

```
askForNewTask=
( (status.countOccupiedMapSlots()<maxMapSlots||
status.countOccupiedReduceSlots()<maxReduceSlots) &&
acceptNewTasks); //存在空闲的Map slot或者Reduce slot，且磁盘剩余空间大于
mapred.local.dir.minspacekill
```

TaskTrackerStatus包含的基本信息如下：

```
String trackerName; //TaskTracker名称
String host; //TaskTracker主机名
int httpPort; //TaskTracker对外的HTTP端口号
int failures; //TaskTracker上已经失败任务总数
List<TaskStatus>taskReports; //当前TaskTracker上各个任务运行状态
volatile long lastSeen; //上次汇报心跳的时间
private int maxMapTasks; //Map slot总数，即允许同时运行的Map Task总数，由参数mapred.
tasktracker.map.tasks.maximum设定
private int maxReduceTasks; //Reduce slot总数，即允许同时运行的Reduce Task总数，由参
数mapred.tasktracker.reduce.tasks.maximum设定
private TaskTrackerHealthStatus healthStatus; //TaskTracker健康状态
private ResourceStatus resStatus; //TaskTracker资源（内存，CPU等）信息
```

下面重点介绍taskReport、healthStatus和resStatus三个变量值的意义及其计算方法。

1.taskReport

taskReport保存当前TaskTracker上所有任务（实际为Task Attempt）的运行状态，每个任务保存信息如下：

```
public abstract class TaskStatus implements Writable, Cloneable{
.....
private final TaskAttemptID taskid; //Task Attempt ID
private float progress; //任务执行进度，范围为0.0~1.0
private volatile State runState; /*任务运行状态，包括RUNNING, SUCCEEDED, FAILED,
UNASSIGNED, KILLED, COMMIT_PENDING, FAILED_UNCLEAN, KILLED_UNCLEAN*/
private String diagnosticInfo; //诊断信息，一般为错误信息和运行异常信息
private String stateString; //字符串形式的运行状态
private String taskTracker; //该TaskTracker名称，可唯一表示一个TaskTracker，形式如
tracker_mymachine: 50010
private int numSlots; //运行该Task Attempt需要的slot数目，默认值是1
private long startTime; //Task Attempt开始时间
private long finishTime; //Task Attempt完成时间
private long outputSize=-1L; //Task Attempt输出数据量
private volatile Phase phase=Phase.STARTING; //任务运行阶段，包括STARTING, MAP,
SHUFFLE, SORT, REDUCE, CLEANUP
private Counters counters; //该任务中定义的所有计数器（包括系统自带计数器和用户自定义计数
器两种）
private boolean includeCounters; //是否包含计数器，计数器信息每隔60 s发送一次
//下一个要处理的数据区间，用于定位坏记录所在区间
private SortedRanges.Range nextRecordRange=new SortedRanges.Range();
.....
}
```

2.healthStatus

healthStatus保存了当前节点健康状况，该变量对应TaskTrackerHealthStatus类，定义如下：

```
static class TaskTrackerHealthStatus implements Writable{
private boolean isNodeHealthy; //节点是否健康
private String healthReport; //如果节点不健康，则记录导致不健康的原因
private long lastReported; //上次汇报健康状况的时间
.....
}
```

healthStatus是由NodeHealthCheckerService线程 [1] 计算得到的。该线程允许管理员配置一个“健康监测脚本”以检查节点健康状况，且管理员可在该脚本中添加任何检查语句作为节点是否健康运行的依据。如果脚本检测到该节点处于不健康状态，它需要在标准输出中打印一条以字符串“ERROR”开头的输出语句。NodeHealthCheckerService线程周期性调用健康监测脚本并检查其输出，一旦发现脚本输出是以“ERROR”开头的字符串，则认为节点处于不健康状态，进而将其标注为“unhealthy”并通过心跳告诉JobTracker，而JobTracker得知节点状态变为“unhealthy”后，会将其加入黑名单，此后不再为它分配新任务。需要注意的是，只要TaskTracker服务是活着的，该线程会一直运行该脚本，一旦发现节点又变为“healthy”，JobTracker会立刻将其从黑名单中移除，从而又会为之分配任务。通过引入该机制，可带来很多好处。

□可作为节点负载的反馈：比如，可让健康检测脚本检查网络、磁盘、文件系统等运行状况，一旦发现特殊情况，比如网络拥塞、磁盘空间不足或者文件系统出现问题，可将健康状况变为“unhealthy”，暂时不接收新的任务，待它们恢复正常后再继续接收新任务。

□人为暂时维护TaskTracker：如果发现TaskTracker所在节点出现故障，可通过控制脚本输出暂时让该TaskTracker停止接收新任务以便进行维护，待维护完成后，修改脚本输出以让TaskTracker继续接收新任务。

NodeHealthCheckerService线程包含四个可配置参数，具体如表7-2所示。用户可根据需要在mapred-site.xml文件中配置这些参数。

表 7-2 NodeHealthCheckerService 线程配置参数

参数名称	参数含义
mapred.healthChecker.script.path	健康监测脚本所在的绝对路径，线程 NodeHealthCheckerService 会周期性执行该脚本以判断节点健康状况，如果该值为空，则不会启动该线程
mapred.healthChecker.interval	健康监测脚本调用频率（单位：毫秒）
mapred.healthChecker.script.timeout	如果健康监测脚本在一定时间内没有响应，则线程 NodeHealthCheckerService 会将节点标注为 “unhealthy”
mapred.healthChecker.script.args	监控脚本的输入参数，如果有多个参数，则用逗号隔开

下面给出一个健康监测脚本实例。在这个Shell脚本中，当一个节点上的空闲内存量低于总内存量的10%时，将打印以“ERROR”开头的字符串，这样，该节点将不再向JobTracker请求新任务。

```
#!/bin/bash
MEMORY_RATIO=0.1
freeMem=grep MemFree/proc/meminfo|awk '{print$2}'
totalMem=grep MemTotal/proc/meminfo|awk '{print$2}'
limitMem=echo|awk '{print int ("'"$totalMem'"*'"$MEMORY_RATIO'"')}'
if[$freeMem-lt$limitMem]; then
echo"ERROR, totalMem=$totalMem, freeMem=$freeMem, limitMem=$limitMem"
else
echo"OK, totalMem=$totalMem, freeMem=$freeMem, limitMem=$limitMem"
fi
```

3.resStatus

resStatus保存了当前TaskTracker资源使用情况。该变量对应ResourceStatus类，定义如下：

```
static class ResourceStatus implements Writable{
private long totalVirtualMemory; //总的可用虚拟内存量，单位为byte
private long totalPhysicalMemory; //总的可用物理内存量
private long mapSlotMemorySizeOnTT; //每个Map slot对应的内存量
private long reduceSlotMemorySizeOnTT; //每个Reduce slot对应的内存量
private long availableSpace; //可用磁盘空间
private long availableVirtualMemory=UNAVAILABLE; //可用的虚拟内存量
private long availablePhysicalMemory=UNAVAILABLE; //可用的物理内存量
private int numProcessors=UNAVAILABLE; //节点总的处理器个数
private long cumulativeCpuTime=UNAVAILABLE; //运行以来累计的CPU使用时间
private long cpuFrequency=UNAVAILABLE; //CPU主频，单位为kHz
```

```
private float cpuUsage=UNAVAILABLE; //CPU使用率，单位为%
.....
}
```

resStatus是由可插拔组件ResourceCalculatorPlugin（抽象类）获取的，当前只存在Linux版本实现的LinuxResourceCalculatorPlugin中，其他操作系统尚未实现，这意味着只有在Linux下才可以获取资源使用信息。此外，Hadoop也允许用户自己编写ResourceCalculatorPlugin实现，且用户只需通过参数mapreduce.tasktracker.resourcecalculatorplugin指定该类即可启用它。

我们都知道，在Linux操作系统中，proc虚拟文件系统中包含了一些目录和文件，它们向用户动态呈现了内核中的一些实时信息，比如进程运行时的资源使用信息。LinuxResourceCalculatorPlugin类正是通过读取/proc目录下的meminfo、cpuinfo和stat三个文件获取节点上的内存、CPU等资源的实时使用情况的。

[1] <https://issues.apache.org/jira/browse/MAPREDUCE-211>

7.3.3 命令执行

JobTracker将心跳应答封装到一个HeartbeatResponse对象中。该对象主要包括两部分内容：第一部分是作业集合recoveredJobs，它是上次关闭JobTracker时正在运行的作业集合，重启JobTracker后需恢复这些作业的运行状态（前提是用户启用了作业恢复功能，具体参考第6章），而TaskTracker收到该作业集合后，需重置这些作业对应Reduce Task的FetchStatus信息，从而迫使这些Reduce Task重新从Map Task端拷贝数据；另一部分是需要执行的命令列表，相关代码如下。

```
TaskTrackerAction[] actions = heartbeatResponse.getActions();
if (reinitTaskTracker(actions)) { //重新初始化
    return State.STALE;
}
if (actions != null) {
    for (TaskTrackerAction action: actions) {
        if (action instanceof LaunchTaskAction) { //启动新任务
            addToTaskQueue((LaunchTaskAction) action);
        } else if (action instanceof CommitTaskAction) { //提交任务
            CommitTaskAction commitAction = (CommitTaskAction) action;
            if (! commitResponses.contains(commitAction.getTaskID())) {
                commitResponses.add(commitAction.getTaskID());
            }
        } else { //杀死任务或者作业
            tasksToCleanup.put(action);
        }
    }
}
```

TaskTracker需要处理5种命令：启动新任务（LaunchTaskAction）、提交任务（CommitTaskAction）、杀死任务（KillTaskAction）、杀死作业（KillJobAction）和重新初始化（ReinitTrackerAction）。我们将在下一节中介绍各个命令的处理过程。

7.4 TaskTracker行为分析

TaskTracker通过心跳机制从JobTracker端获取各种命令，包括启动新任务、提交任务、杀死任务、杀死作业和重新初始化等。在Hadoop中，存在多种场景使得JobTracker下达这些命令。比如，对于“杀死任务”命令而言，可能是人为通过Shell命令杀死任务，也可能由于任务超量使用内存，由框架直接将其杀死。在本节中，我们为每个命令选取一种场景来讲解其执行全过程。

7.4.1 启动新任务

TaskTracker出现空闲资源后，会通过心跳从JobTracker端索取任务，并按照一定步骤启动该任务，之后一直监控并汇报其运行状态，直到它运行成功。新任务启动过程如图7-3所示。

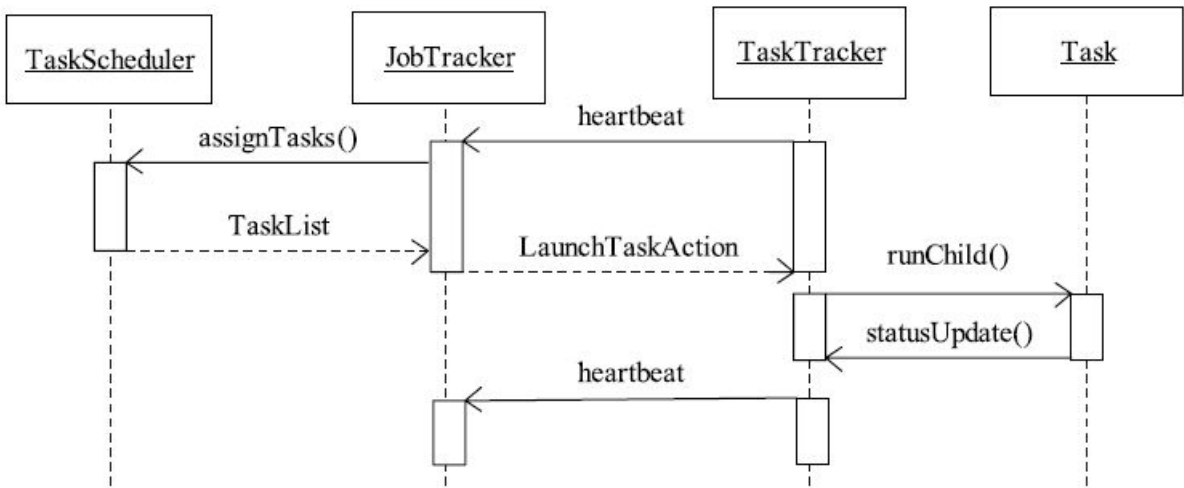


图 7-3 新任务启动序列图

- 1) TaskTracker出现空闲资源，将资源使用情况通过心跳汇报给JobTracker;
- 2) JobTracker收到信息后，解析心跳信息，发现TaskTracker上有空闲的资源，则调用调度器模块的assignTasks()函数为该TaskTracker分配任务;
- 3) JobTracker将新分配的任务封装成一个或多个LaunchTaskAction对象，将其添加到心跳应答中返回给TaskTracker;
- 4) TaskTracker收到心跳后，解析出LaunchTaskAction对象，并创建JVM启动任务;
- 5) 当Counter值或者进度发生变化时，任务通过RPC函数将最新Counter值和进度汇报给TaskTracker;
- 6) TaskTracker通过心跳将任务的最新Counter值和进度汇报给JobTracker。

我们将在7.5节详细剖析TaskTracker启动新任务的整个过程。

7.4.2 提交任务

任务提交过程是指任务处理完数据后，将最终计算结果从临时目录转移到最终目录的过程。需要注意的是，只有将输出结果直接写到HDFS上的任务才会经历该过程。在Hadoop中，有两种这样的任务：**Reduce Task**和**map-only**类型作业的**Map Task**。

前面提到Hadoop的推测执行机制：**Hadoop**允许多个任务同时处理同一份数据，但只会选择最先运行完成的任务处理结果作为最终结果。为了防止多个任务产生相同的处理结果而造成冗余，每个任务暂时将自己的计算结果放到一个临时目录中，一旦处理完后，先向**JobTracker**发送结果提交请求，得到**JobTracker**准许后，才可以将结果从临时目录转移到最终目录中，而一旦一个任务提交结果后，其计算结果便会作为最终结果，其他任务的计算结果将被丢弃。

Hadoop任务提交过程采用了两阶段提交协议（**two-phase commit protocol, 2PC**）实现。两阶段提交协议是分布式事务中经常采用的协议。它把分布式事务的某一个代理指定为协调者，所有其他代理称为参与者，同时规定只有协调者才有提交或撤销事务的决定权，而其他参与者各自负责在其本地执行写操作，并向协调者提出撤销或提交子事务的意向。两阶段提交协议把事务提交分成两个阶段。

- 第一阶段（准备阶段）：各个参与者执行完自己的操作后，将状态变为“可以提交”，并向协调者发送“准备提交”请求。
- 第二阶段（提交阶段）：协调者按照一定原则决定是否允许参与者提交，如果允许，则向参与者发出“确认提交”请求，参与者收到请求后，把“可以提交”状态改为“提交完成”状态，然后返回回答；如果不允许，则向参与者发送“提交失败”请求，参与者收到该请求后，把“可以提交”状态改为“提交失败”状态，然后退出。

对于**MapReduce**而言，**JobTracker**扮演协调者的角色，而各个**TaskTracker**上的任务是参与者。任务提交过程对应的两阶段提交协议实现如图7-4所示。

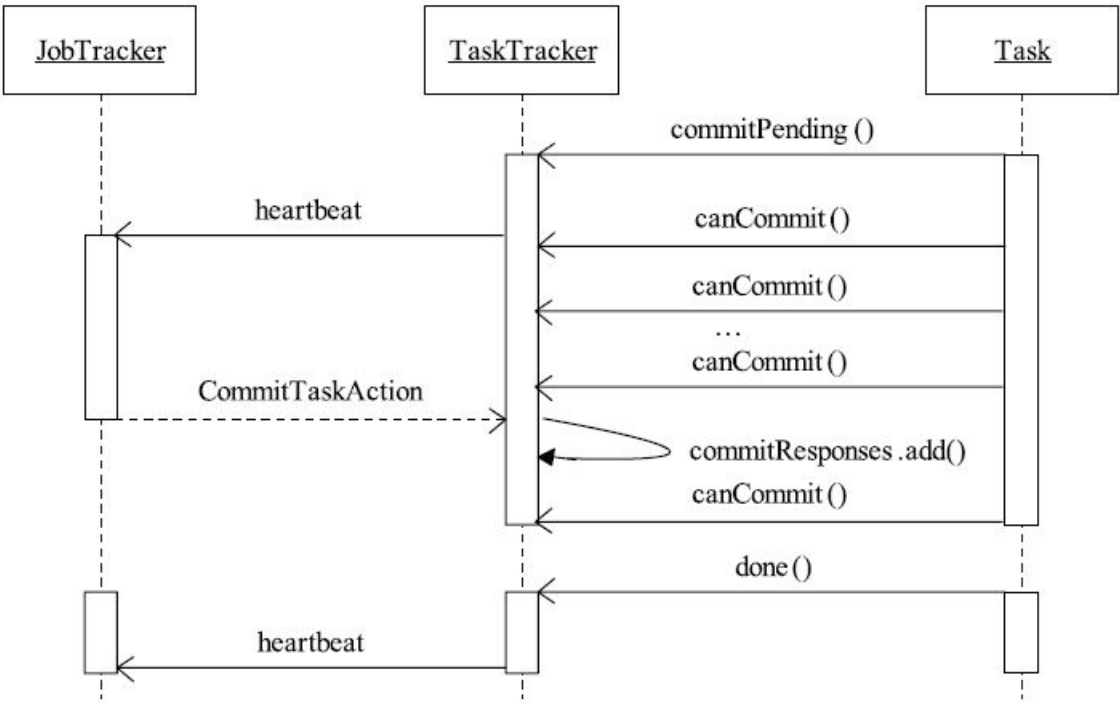


图 7-4 任务提交序列图

- 1) **Task Attempt**处理完最后一条记录后（此时数据写在临时目录`${mapred.output.dir}/_temporary/_${taskid}`下），运行状态由**RUNNING**变为**COMMIT_PENDING**，并通过RPC将该状态和任务提交请求发送给**TaskTracker**；
- 2) **TaskTracker**得知一个**Task Attempt**状态为**COMMIT_PENDING**后，立刻缩短心跳间隔，以便快速将任务状态汇报给

JobTracker;

3) JobTracker收到心跳信息后, 检查它是否为某个TaskInProgress中第一个状态变为COMMIT_PENDING的Task Attempt, 如果是, 则批准它进行任务提交, 即在心跳应答中添加CommitTaskAction命令, 以通知TaskTracker准许该任务提交最终结果;

4) TaskTracker收到CommitTaskAction命令后, 将对应任务加入可提交任务列表commitResponses中;

5) Task Attempt通过RPC检测到自己位于列表commitResponses中后, 进行结果提交, 即将数据从临时目录转移到最终目录(参数\${mapred.output.dir}对应的目录)中, 并告诉TaskTracker任务提交完成;

6) TaskTracker得知任务提交完成后, 将该任务运行状态由COMMIT_PENDING变为SUCCEEDED, 并在下次心跳中将该任务状态汇报给JobTracker。

综合以上几个步骤, 并结合2PC的定义, 很容易知道: 上面第1步对应2PC的第一阶段, 而其余各步骤对应2PC的第二阶段。

7.4.3 杀死任务

Hadoop中存在多种场景将一个任务杀死，它们涉及的过程基本相同，均是通过JobTracker向TaskTracker发送KillTaskAction命令完成的。本小节分析用户使用Shell命令杀死任务的整个过程，具体如图7-5所示。

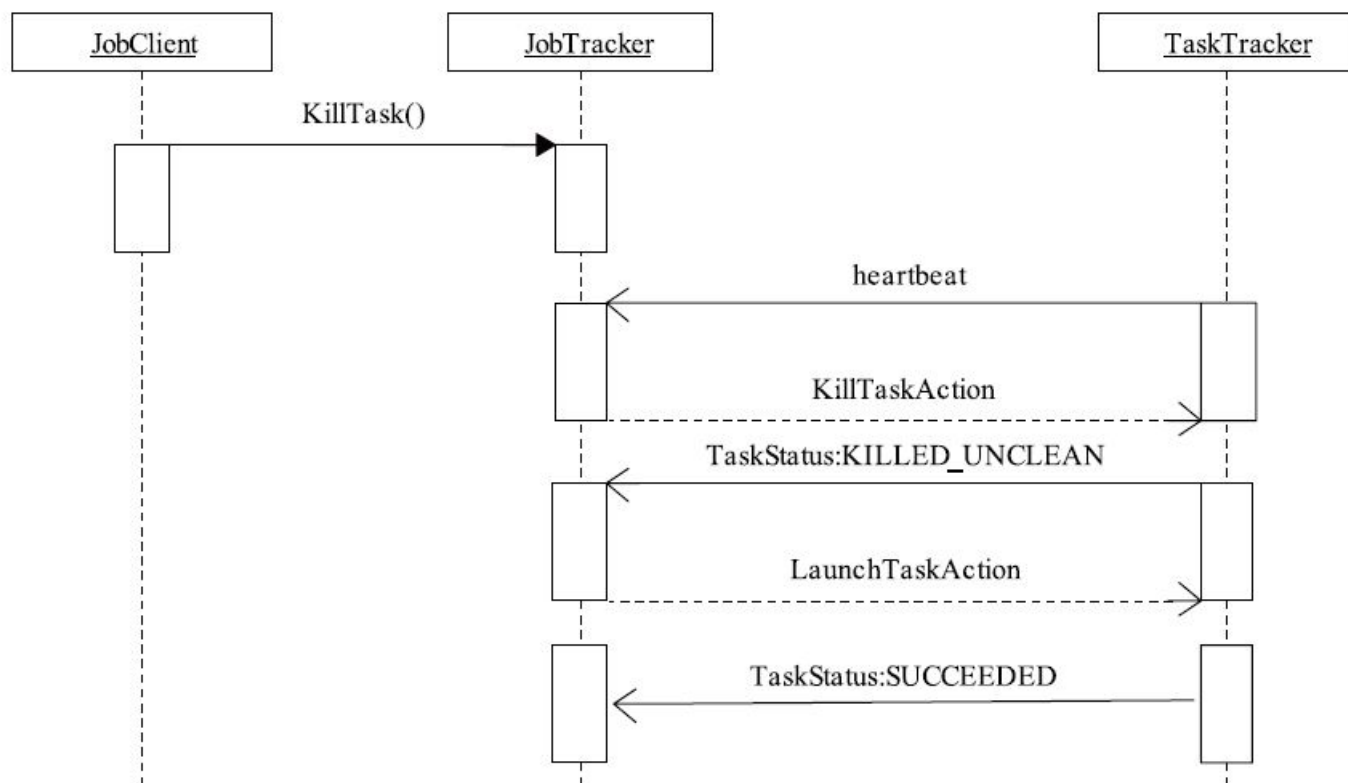


图 7-5 杀死任务序列图

用户输入“bin/hadoop job-kill-task<task-attempt-id>”后，JobClient内部调用RPC函数KillTask杀死该Task Attempt，整个过程如下：

- 1) JobTracker收到来自JobClient的杀死任务请求后，将该任务添加到待杀死任务列表tasksToKill中。
- 2) 之后某个时刻，Task Attempt所在的TaskTracker向JobTracker发送心跳，JobTracker收到心跳后，将该任务封装到KillTaskAction命令中，并通过心跳应答发送给TaskTracker。
- 3) TaskTracker收到KillTaskAction命令后，将该任务从作业列表runningJobs中清除，并将运行状态从RUNNING转化为KILLED_UNCLEAN，同时通知directoryCleanupThread线程清理其工作目录，释放所占槽位，最后缩短心跳时间间隔，以便将该任务状态迅速通过带外心跳告诉JobTracker。
- 4) JobTracker收到状态为KILLED_UNCLEAN的Task Attempt后，将其类型改为task-cleanup task（这种任务的输入数据为空，但ID与被杀Task Attempt的ID相同，其目的是清理被杀Task Attempt已经写入HDFS的临时数据），并添加到待清理任务队列mapCleanupTasks/reduceCleanupTasks中，在接下来的某个TaskTracker的心跳中，JobTracker将其封装到LaunchTaskAction中发送给TaskTracker。
- 5) TaskTracker收到LaunchTaskAction后，启动JVM（或重用已启动的JVM）执行该任务。由于该任务属于task-cleanup task，因此它只需清理被杀死的Task Attempt已写入HDFS的临时数据（如果没有则直接跳过），之后其运行状态变为SUCCEEDED，并由TaskTracker通过下一次心跳告诉JobTracker。

6) JobTracker收到运行状态为SUCCEDED的Task Attempt后，首先检查它是否位于任务列表tasksToKill中，显然该任务在该列表中，这表明它已经被杀死，于是将其状态转化为KILLED，同时修改相应的各个数据结构。

从上面整个过程可以看出，JobClient向JobTracker发出“kill task”请求后，JobTracker不会返回任何确认消息。这主要是由于杀死任务的过程比较复杂，要经历多个心跳时间，JobClient需等待很长时间才可能知道任务是否被成功杀死。

7.4.4 杀死作业

杀死作业是通过JobTracker向TaskTracker发送KillTaskAction命令完成的。它是Hadoop最常见的行为之一，比如，任何一个作业成功运行完成后，JobTracker会向各个TaskTracker广播KillJobAction以清空各个节点上该作业的工作目录。此外，用户可以通过调用Shell命令“bin/hadoop job-kill<job-id>”杀死一个作业。本小节主要分析用户使用Shell命令杀死作业的过程，具体如图7-6所示。

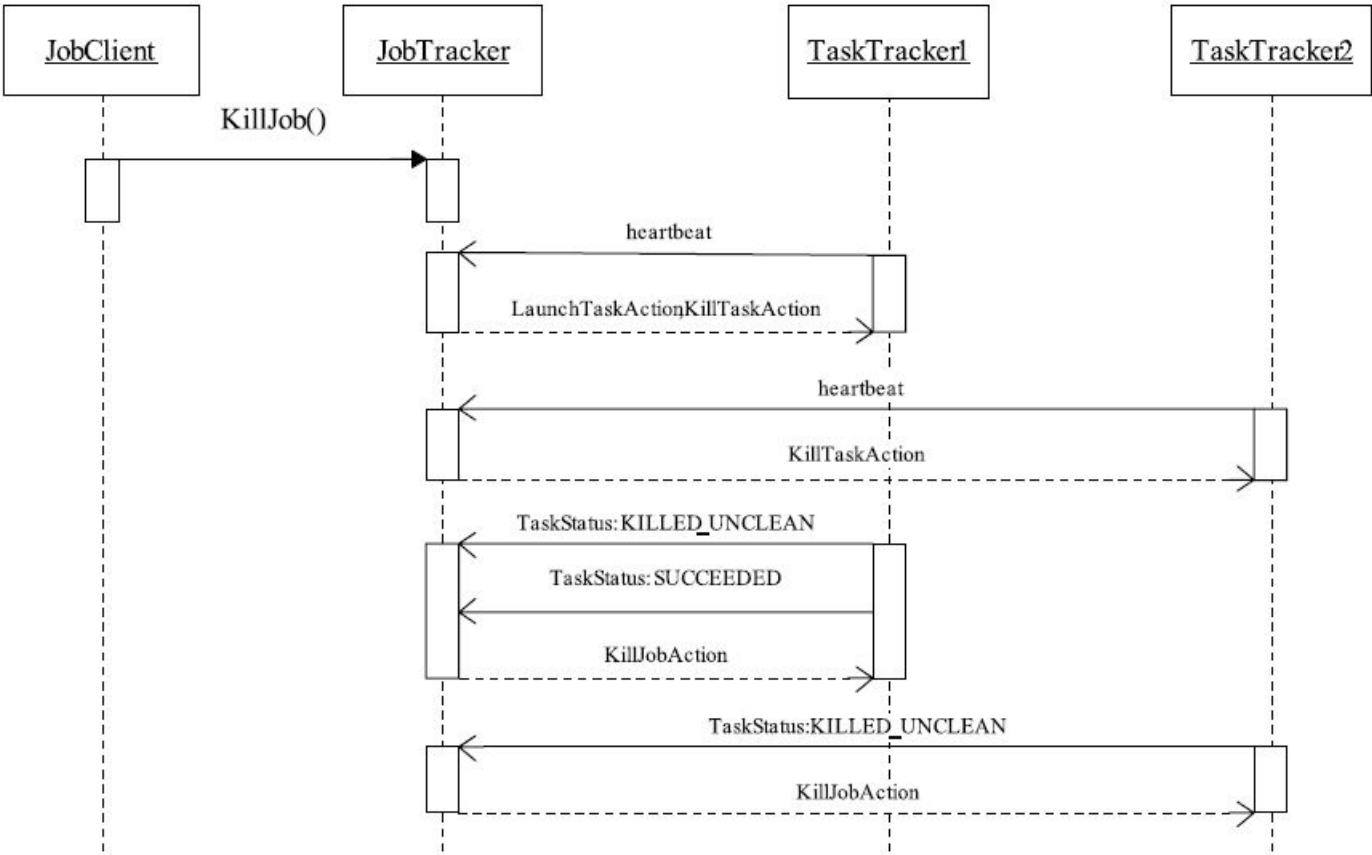


图 7-6 杀死作业序列图

用户输入一条“杀死任务”的Shell命令后，JobClient通过RPC函数KillJob向JobTracker发送杀死作业请求。JobTracker收到请求后，涉及的过程如下：

- 1) 作业的JobInProgress对象将jobKilled变量置为true，并杀死该作业所有的job-setup task、map task和reduce task（注意：job-cleanup task保留）。
- 2) 此后假设TaskTracker1第一个向JobTracker汇报心跳，则JobTracker将该作业的job-cleanup task封装成LaunchTaskAction命令，将该TaskTracker对应的已被杀任务封装成KillTaskAction命令，并把这些命令添加到心跳应答中返回给TaskTracker1。
- 3) 对于接下来发送心跳的TaskTracker, JobTracker将对应的被杀死任务封装成KillTaskAction命令，并把这些命令添加到心跳应答中返回给这些TaskTracker。
- 4) TaskTracker1收到来自JobTracker的命令后，执行相应的操作：若为KillTaskAction命令，则进行的操作与“杀死任务”中的第3步类似，执行完命令后，将作业状态KILLED_UNCLEAN汇报给JobTracker；若为LaunchTaskAction命令，TaskTracker将创建JVM执行该任务，但由于该任务没有需要处理的数据，因此很快可以运行完成，任务完成后，其状态转化为SUCCEEDED，并由TaskTracker通过心跳告诉JobTracker。

5) 其他TaskTracker收到JobTracker端的KillTaskAction命令后，进行与第4步类似的操作。

6) JobTracker收到TaskTracker汇报的心跳后，若心跳信息包含任务状态变为KILLED_UNCLEAN，由于任务所属作业已被杀死（jobKilled=true），则将任务由KILLED_UNCLEAN状态转化为KILLED状态；若心跳信息包含job-cleanup task运行成功（状态为SUCCEEDED），则向各个TaskTracker广播KillJobAction命令，以清理作业的工作目录和相关的内存结构（比如runningJobs）。

7.4.5 重新初始化

如果TaskTracker上某个磁盘出现故障或者TaskTracker收到来自JobTracker的ReinitTrackerAction命令，则TaskTracker会重新进行初始化。重新初始化过程与TaskTracker启动过程是一致的，可参考7.2节。

7.5 作业目录管理

在MapReduce计算过程中，Map Task要将大量中间数据写入本地磁盘，而这些数据不存在备份，一旦丢失后，就必须重新计算。为了提高这部分数据的可靠性和并发写性能，Hadoop允许TaskTracker配置多个挂在不同磁盘的目录作为中间结果存放目录。对于任意一个作业，Hadoop会在每个磁盘创建相同的目录结构，然后采用轮询策略使用这些目录（由类LocalDirAllocator实现）。

TaskTracker上的目录可分为两种：数据目录和日志目录。其中，数据目录用于存放执行任务所必须的数据（比如可执行程序或jar包，作业配置文件等）和运行过程中产生的临时数据，由参数mapred.local.dir指定；而日志目录则用于存放TaskTracker和Task运行时输出日志，由参数hadoop.log.dir指定。下面我们分别介绍这两种目录的组织方式。

1.数据目录

假设某个TaskTracker上通过参数mapred.local.dir配置了N个目录/mnt/disk0, /mnt/disk1,, /mnt/diskN-1, 且这N个目录正好挂在了N个不同的磁盘，某时刻用户提交了一个ID为jobid1的作业，该作业包含K个任务（为简化说明，在此不区分任务类型），则TaskTracker为该作业创建的目录结构如图7-7所示。TaskTracker在每个磁盘上为该作业创建了相同的目录结构，且采用轮询的方式使用这些目录。比如，对于任务taskid1，它需要创建工作目录work和输出结果目录output，为了分摊写负载，TaskTracker可能将work目录放到/mnt/disk1/磁盘上，而将output目录放到/mnt/disk2磁盘上。

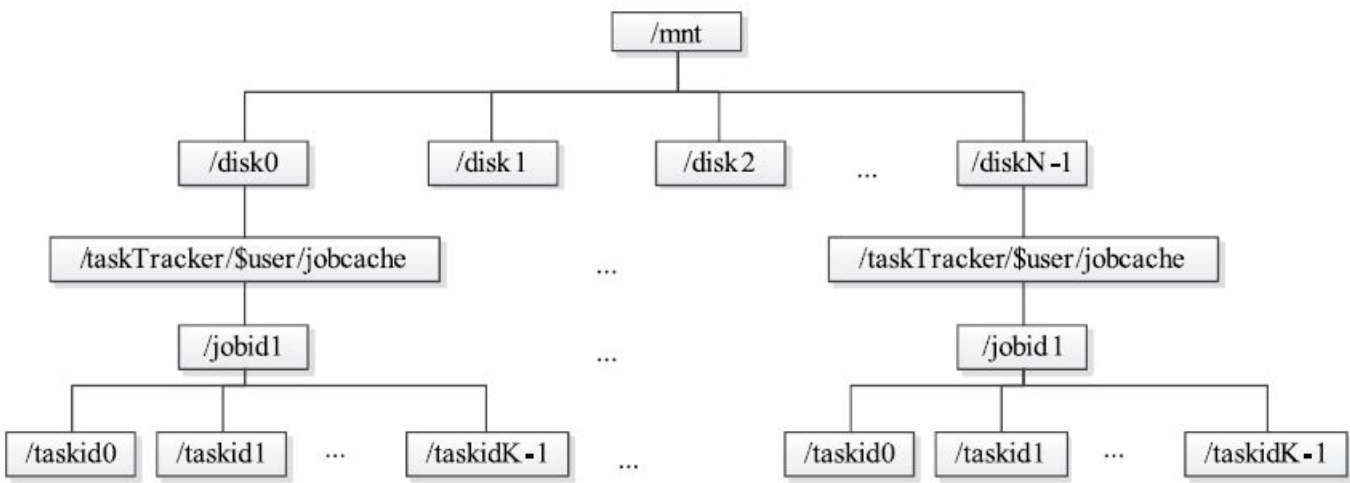


图 7-7 作业数据目录结构

考虑到Reduce Task可能会运行失败，且每个Reduce Task要从所有Map Task中获取部分输入，因此，所有任务目录不会在作业运行过程中被删除，而是确认作业运行完成后，统一将其删除。

2.日志目录

(1) 日志目录创建

不同于数据目录，Hadoop只允许TaskTracker将日志目录存在一个磁盘上。一个典型的日志目录结构如图7-8所示。TaskTracker包含两种日志：系统日志和用户日志。其中，系统日志是TaskTracker服务内部打印的运行日志，存放到文件<tasktracker-name>.log和<tasktracker-name>.out中；而用户日志存放在userlogs目录下，且按照不同作业不同任务分别建立子目录。

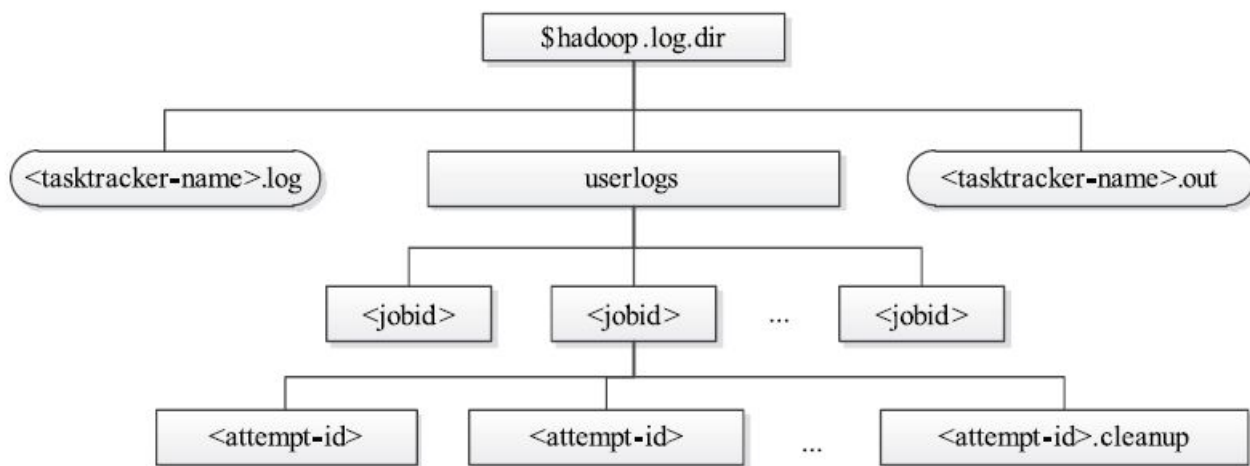


图 7-8 作业日志目录结构

如果TaskTracker上任务多、日志量大，则这种日志目录组织方式可能成为TaskTracker的性能瓶颈。为此，从Hadoop 0.20.204.0开始引入了多磁盘日志目录组织方式：系统日志仍被直接放到hadoop.log.dir目录下，但用户日志将被采用轮询的方式将分布到mapred.local.dir指定的各个磁盘目录下，同时为了保持语义不变，TaskTracker在hadoop.log.dir目录下创建指向这些目录的符号链接，具体如图7-9所示。

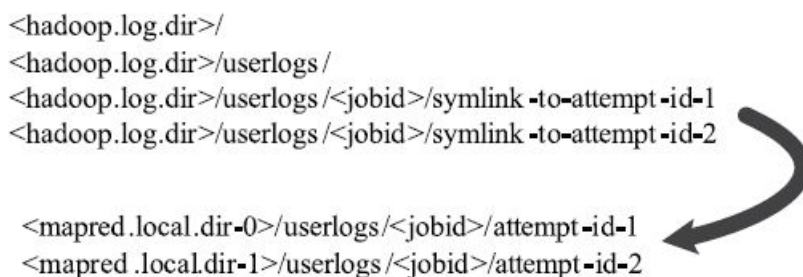


图 7-9 多磁盘作业日志目录结构

这种用户日志目录组织方式带来的好处如下：

- 可以高效应对更大量的用户日志，且TaskTracker不再被单磁盘日志读写性能约束，同时更可靠；
- 在修改很少代码的前提下，维持了之前的语义。

(2) 日志目录清理

日志目录清理是由类UserLogManager实现的。它包含两个非常重要的成员变量：taskLogsTruncater和userLogCleaner，它们分别用于日志裁剪和日志清理，具体如下。

□taskLogsTruncater：用户输出的日志文件可能很大，这将占用大量磁盘空间，同时对前端日志信息展示也不够友好，为此，TaskTracker通过TaskLogsTruncater类实现了日志文件裁剪功能，即当任务运行完成后，TaskTracker将按照管理员要求对各个日志文件进行裁剪，以保证日志文件不会太大。TaskTracker为管理员提供了两个可配置参数：mapreduce.cluster.map.userlog.retain-size和mapreduce.cluster.reduce.userlog.retain-size，分别用于配置Map Task和Reduce Task最大可保留的日志文件大小。

□userLogCleaner：由于TaskTracker将所有作业的运行日志保存到本地磁盘上，因此随着时间的积累，作业日志必将越来越多。为了避免作业日志占用大量磁盘空间，TaskTracker通过userLogCleaner线程实现了日志文件清理功能。TaskTracker允许一个作业日志可在磁盘上的保留时间为mapred.userlog.retain.hours（默认为24小时），一旦超过该时间，TaskTracker会将该作业日志从磁盘上删除。

7.6 启动新任务

TaskTracker最重要的任务之一是启动JobTracker分配的新任务并周期性汇报它们的运行状态。一个任务的启动过程如图7-10所示，大致经历两个步骤：作业本地化和启动任务（包括任务本地化和运行任务）。

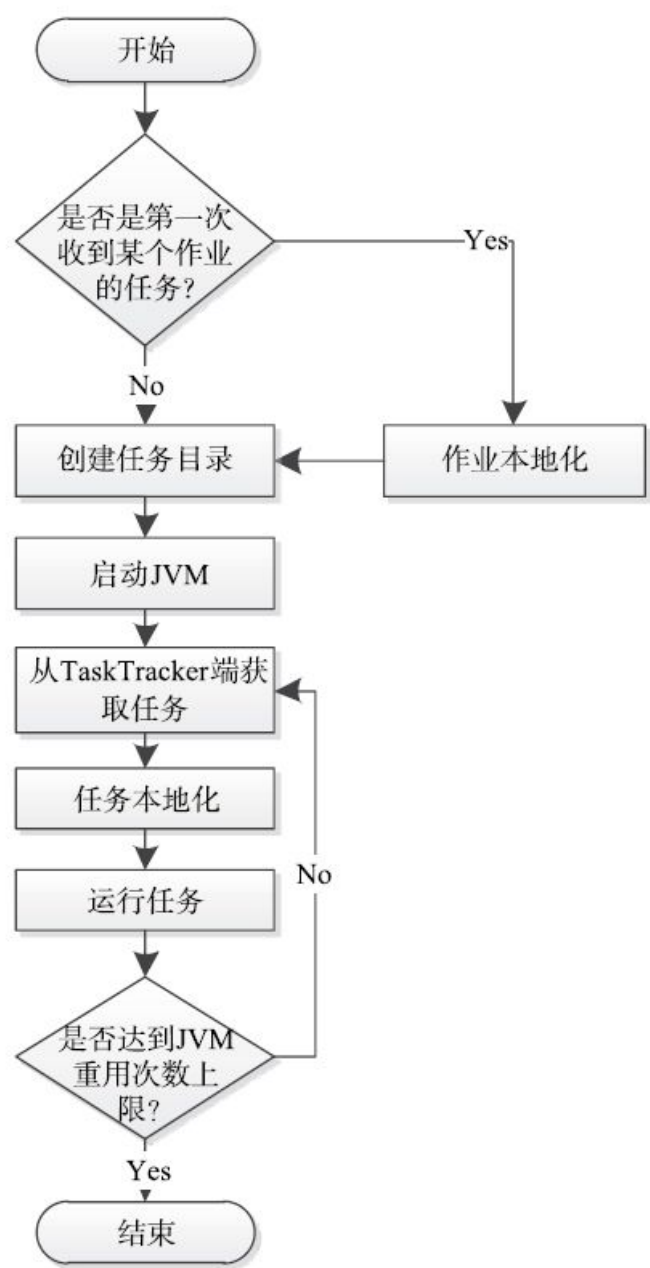


图 7-10 TaskTracker任务启动流程图

7.6.1 任务启动过程分析

1.作业本地化

本地化（localize）是指在TaskTracker上为作业和任务构造一个运行环境，包括创建作业和任务的工作目录，（从HDFS上）下载运行任务相关的文件，如程序jar包、字典文件等，设置环境变量等，可分为作业本地化和任务本地化。在同一个TaskTracker上，由作业的第一个任务完成该作业的本地化工作，后续任务只需进行任务本地化。

由于用户应用程序相关文件（比如jar包，字典文件等）可能很大，这使得任务花费较长时间从HDFS上下载这些数据，如果TaskTracker串行启动各个任务，势必会延长任务的启动时间。为了解决该问题，TaskTracker采用多线程启动任务，即为每个任务

单独启动一个线程：

```
void startNewTask (final TaskInProgress tip) {
    Thread launchThread=new Thread (new Runnable() {
    public void run(){
    .....
    RunningJob rjob=localizeJob (tip); //作业本地化
    tip.getTask().setJobFile (rjob.getLocalizedJobConf().toString());
    launchTaskForJob (tip, new JobConf (rjob.getJobConf()), rjob);
    .....
    }
    }
}
```

为了防止同一个作业的多个任务同时进行作业本地化，TaskTracker需要对相关数据结构加锁。一种常见的加锁方法是：

```
RunningJob rjob;
.....//对rjob赋值
synchronized (rjob) { //获取rjob锁
    initializeJob (rjob); /*作业本地化。如果作业文件很大，则该函数运行时间很长，这导致rjob锁
    长时间不释放*/
}
```

在以上实现方案中，作业本地化过程会一直占用rjob锁，这导致很多其他需要该锁的线程或者函数不得不等待，如MapEventsFetcherThread线程、TaskTracker.getMap-CompletionEvents()函数等。

为了避免作业本地化过程中长时间占用rjob锁，TaskTracker为每个正在运行的作业维护两个变量：localizing和localized，分别表示正在进行作业本地化和已经完成作业本地化。通过对这两个变量的控制可避免作业本地化时对RunningJob对象加锁，且能够保证只有作业的第一个任务进行作业本地化：

```
synchronized (rjob) {
    if (! rjob.localized) { //该作业尚未完成本地化工作
        while (rjob.localizing) { //另外一个任务正在进行作业本地化
            rjob.wait(); //等待作业本地化结束
        } //释放rjob锁
        if (! rjob.localized) { //没有任务进行作业本地化
            rjob.localizing=true; //让当前任务对该作业进行本地化
        }
    }
    if (! rjob.localized) { //运行到此，说明当前没有任务进行作业本地化
        initializeJob (rjob); //进行作业本地化工作
    }
}
```

在整个作业本地化过程中，<localizing, localized>两个变量值变化过程为：

```
<false, false>→<true, false>→<true, true>
```

作业的第一个任务负责为该作业本地化，具体步骤如下：

步骤1 将凭据文件jobToken和作业描述文件job.xml下载到TaskTracker私有文件目录中。

TaskTracker私有文件目录是\${mapred.local.dir}/ttprivate，其中，不同用户的文件放到不同子目录下，比如用户\$user提交的作业\$jobid对应文件放在\${mapred.local.dir}/ttprivate/taskTracker/\$user/jobcache/\$jobid/目录下。此外，在任务运行过程中，任务启动脚本taskjvm.sh（具体见下一小节）也存放在该目录下。

步骤2 将其他初始化工作交给TaskController.initializeJob函数处理。

TaskController类主要用于控制任务的初始化、终结和清理等工作，当前默认实现是DefaultTaskController。TaskController.initializeJob函数的主要工作是创建作业相关目录和文件，最终生成的目录结构如下。

□ \${mapred.local.dir}/taskTracker/distcache/：TaskTracker上的public级别分布式缓存，该TaskTracker上所有用户的所有作业共享该缓存中的文件。

□ `${mapred.local.dir}/taskTracker/$user/distcache/`: TaskTracker上的private级别缓存, 用户\$用er的所有作业共享该缓存中的文件。

□ `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/`: 用户\$用er提交的作业\$jobid对应的目录。

□ `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/work/`: 作业共享目录, 可作为任务的数据暂存目录或者共享目录, 可通过`JobConf.getJobLocalDir()`函数获取。此外, 它还在系统属性中, 因此也可以通过`System.getProperty("job.local.dir")`获取。

□ `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/jars/`: 存放作业jar文件和展开后的jar文件。程序相关的jar文件被统一命名成`job.jar`, 并分发到各个节点上, 且在任务运行之前, 由Hadoop自动展开该文件。

□ `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/job.xml`: 存放作业相关的配置属性。

□ `$mapred.local.dir/taskTracker/$user/jobcache/$jobid/jobToken`: 作业凭据文件, 用于保证作业运行安全。

□ `$mapred.local.dir/taskTracker/$user/jobcache/$jobid/job-acls.xml`: 保存作业访问控制权限。

□ `$hadoop.log.dir/userlogs/$jobid/`: 作业日志存放目录。

2.启动任务

为了避免不同任务之间相互干扰, TaskTracker为各个任务启动了独立的JVM。也就是说, JVM相当于包含一定资源量的容器, 每个任务可在该容器使用其资源运行。这里先介绍JVM启动过程, 然后介绍任务启动过程。

根据任务类型, TaskTracker会调用不同的TaskRunner启动任务。对于Map Task, 会调用`MapTaskRunner`, 而Reduce Task则调用`ReduceTaskRunner`, 但任务启动最终均是由`TaskRunner.run()`方法完成的。

`TaskRunner.run()`方法首先准备启动任务需要的各种信息, 包括启动命令、启动参数、环境变量、标准输出流、标准错误输出流等信息, 然后交给`JvmManager`对象启动一个JVM。

`JvmManager`负责管理TaskTracker上所有正在使用的JVM, 包括启动、停止、杀死JVM等。考虑到一般情况下Map Task和Reduce Task占用的资源量不同, `JvmManager`使用`mapJvmManager`和`reduceJvmManager`单独管理两种类型任务对应的JVM, 且规定:

□ 每个TaskTracker上同时启动的Map Task和Reduce Task数目不能超过Map slot和Reduce slot数目;

□ 每个JVM只能同时运行一个任务;

□ 每个JVM可重复使用以减少启动开销(重用次数可通过参数`mapred.job.reuse.jvm.num.tasks`指定), 但某个JVM只限于同一个作业的同类型任务使用。这一点可从JVM ID中看出, 它是某个作业ID(将其ID标识字符串“job”变为“jvm”), 任务类型和一个随机整型拼接而成的, 比如`jvm_201209031104_0010_m_482270223`。

JVM启动过程如下:

步骤1 如果已启动JVM数目低于上限数目(Map slot或者Reduce slot数目), 则直接启动JVM, 否则进入步骤2。

步骤2 查找当前TaskTracker所有已经启动的JVM, 找出满足以下条件的JVM:

□ 当前状态为空闲;

□ 复用次数未超过上限数目;

□与将要启动的任务同属一个作业（通过JvmID可获取作业ID）。

如果找到这样的JVM，则可继续复用而无须启动新的JVM，否则进入步骤3。

步骤3 查找当前TaskTracker所有已经启动的JVM，如果满足以下两个条件之一，则直接将该JVM杀掉，并启动一个新的JVM。

□复用次数已达到上限数目且与新任务同属一个作业；

□当前处于空闲状态但与新任务不属于同一作业。

启动JVM是由JvmRunner线程完成的，它进一步调用了TaskController中的launchTask方法。在DefaultTaskController实现中，该方法首先在本地的磁盘创建任务工作目录，接着将任务启动命令写到Shell脚本taskjvm.sh中，并直接使用以下命令运行该脚本以启动任务：

```
bash-c taskjvm.sh
```

其中，一个taskjvm.sh实例如下：

```
export JVM_PID=echo$$
export HADOOP_CLIENT_OPTS="-Dhadoop.tasklog.taskid=attempt_201209010905_0002_m_000000_0-Dhadoop.tasklog.iscleanup=false-Dhadoop.tasklog.totalLogFileSize=0"
export SHELL="/bin/bash"
export HADOOP_WORK_DIR="/tmp/mapred/taskTracker/intuser/jobcache/job_201209010905_0002/attempt_201209010905_0002_m_000000_0/work"
export HOME="/homes/"
export LOGNAME="dongxicheng"
export HADOOP_TOKEN_FILE_LOCATION="/tmp/mapred/taskTracker/intuser/jobcache/job_201209010905_0002/jobToken"
export HADOOP_ROOT_LOGGER="INFO, TLA"
export LD_LIBRARY_PATH="/tmp/mapred/taskTracker/intuser/jobcache/job_201209010905_0002/attempt_201209010905_0002_m_000000_0/work:/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/jre/lib/amd64/server:/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/jre/lib/amd64:/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/jre/./lib/amd64"
export USER="dongxicheng"
exec setsid '/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/jre/bin/java' .....
'-Dhadoop.log.dir=/home/dongxicheng/hadoop-1.0.0/libexec/./logs'
'-Dhadoop.root.logger=INFO, TLA'
'-Dhadoop.tasklog.taskid=attempt_201209010905_0002_m_000000_0'
'-Dhadoop.tasklog.iscleanup=false'-Dhadoop.tasklog.totalLogFileSize=0'
'org.apache.hadoop.mapred.Child'127.0.0.1'47655'attempt_201209010905_0002_m_000000_0'/home/dongxicheng/hadoop-1.0.0/libexec/./logs/userlogs/job_201209010905_0002/attempt_201209010905_0002_m_000000_0'400940185'</dev/null 1>>/home/dongxicheng/hadoop-1.0.0/libexec/./logs/userlogs/job_201209010905_0002/attempt_201209010905_0002_m_000000_0/stdout2>>/home/dongxicheng/hadoop-1.0.0/libexec/./logs/userlogs/job_201209010905_0002/attempt_201209010905_0002_m_000000_0/stderr
```

可以看到，最终是通过org.apache.hadoop.mapred.Child类运行任务的，即

```
org.apache.hadoop.mapred.Child<host><port><task-attempt-id><log-location><jvm-id>
```

上面代码中的五个输入参数分别表示TaskTracker的IP、端口号、Task Attempt ID、日志位置和JVM ID。

其中，org.apache.hadoop.mapred.Child类的核心代码框架如下：

```
public static void main (String[] args) throws Throwable{
//创建RPC Client，启动日志同步线程
.....
while (true) { //不断询问TaskTracker，以获取新任务
JvmTask myTask=umbilical.getTask(context); //获取新任务
if (myTask.shouldDie()) { //JVM所属作业不存在或者被杀死
break;
}else{
if (myTask.getTask()==null) { //暂时没有新任务
//等待一段时间继续询问TaskTracker
.....
}
```

```
continue;
}
}
//有新任务，进行任务本地化
.....
taskFinal.run (job, umbilical); //启动该任务
.....
//如果JVM复用次数达到上限数目，则直接退出
if (numTasksToExecute>0&&++numTasksExecuted==numTasksToExecute) {
break;
}
}
}
```

其中任务本地化涉及内容如下：

1) 将任务相关的一些配置参数添加到作业配置JobConf中（如果参数名相同，则会覆盖），形成任务自己的配置JobConf，并采用轮询的方式选择一个目录存放对应的任务对象配置文件。也就是说，任务的JobConf是由作业的JobConf与任务特定参数组成的，其中，任务特定参数如表7-3所示。

表 7-3 任务特定参数列表

参数名称	类型	参数含义
mapred.job.id	String	作业 ID
mapred.jar	String	作业的 job.jar 所在目录
job.local.dir	String	作业共享目录
mapred.tip.id	String	任务 ID
mapred.task.id	String	Task Attempt ID
mapred.task.is.map	boolean	是否为 Map Task
mapred.task.partition	int	任务在作业中的 ID
map.input.file	String	Map Task 的输入文件
map.input.start	long	Map Task 的输入数据在输入文件中的偏移量
map.input.length	long	Map 输入 InputSplit 的长度

2) 在工作目录中建立指向分布式缓存中所有数据文件的链接，以便能够直接使用这些文件。

最终，形成的任务目录结构如下。

- \${mapred.local.dir}/taskTracker/\$user/jobcache/\$jobid/\$taskid: 作业\$jobid中的任务\$taskid对应的目录。
- \${mapred.local.dir}/taskTracker/\$user/jobcache/\$jobid/\$taskid/job.xml: 任务本地化后产生的与该任务对应的配置文件。
- \${mapred.local.dir}/taskTracker/\$user/jobcache/\$jobid/\$taskid/split.info: 任务的InputSplit元数据信息，仅用于IsolationRunner调试。
- \${mapred.local.dir}/taskTracker/\$user/jobcache/\$jobid/\$taskid/output: 存放中间输出文件的目录，比如Map Task的中间输出结果。
- \${mapred.local.dir}/taskTracker/\$user/jobcache/\$jobid/\$taskid/work: 任务的工作目录。
- \${mapred.local.dir}/taskTracker/\$user/jobcache/\$jobid/\$taskid/work/tmp: 任务的临时目录。用户可通过参数mapred.child.tmp设置Map Task和Reduce Task的临时目录，默认值是./tmp。如果该值不是绝对路径，则任务是相对于工作目录的相对路径。当启动JVM时，会将-Djava.io.tmpdir='the tmp dir'作为启动参数。
- \$hadoop.log.dir/userlogs/\$jobid/\$taskid: 作业\$jobid中任务\$taskid的日志目录。该目录下的主要文件或子目录如下。

○\$Hadoop.log.dir/userlogs/\$jobid/\$taskid/stdout: 应用程序打印的标准输出数据, 比如Java应用程序中使用System.out.print()函数打印的输出数据。

○\$Hadoop.log.dir/userlogs/\$jobid/\$taskid/stderr: 应用程序打印的标准错误输出数据, 比如Java应用程序中使用System.err.print()函数打印的输出数据。

○\$Hadoop.log.dir/userlogs/\$jobid/\$taskid/syslog: 应用程序和MapReduce框架打印的系统日志, 比如应用程序中使用LOG (INFO) 打印的日志。

○\$Hadoop.log.dir/userlogs/\$jobid/\$taskid/profile.out: profiling日志。当用户设置配置选项mapred.task.profile=true时, TaskTracker会启用任务的profiling功能, 这会根据用户要求采集任务的运行时信息并保存到profile.out文件中, 以方便用户对应用程序调优。

○\$Hadoop.log.dir/userlogs/\$jobid/\$taskid/debugout: Debug脚本的标准输出。用户可通过参数mapred.map.task.debug.script和mapred.reduce.task.debug.script指定任务失败时需运行的调试脚本。

Debug脚本的标准输出举例如下:

步骤1 编写应用程序。

为了简化问题, 我们采用Hadoop Streaming编写应用程序。为了能够让任务运行失败, 我们在Map Task处理第5行数据记录时, 打印一个空指针, 对应的C++代码如下:

```
#include<string>
#include<iostream>
using namespace std;
int main(){
string key;
int errorno=1;
while (cin>>key) {
cout<<key<<"\t"<<"1"<<endl;
if (errorno++==5) {
int*p=NULL;
cout<<*p<<endl; //当处理到第5行记录时, 让Map Task运行失败
}
}
return 0;
}
```

编译以上程序生成可执行程序Mapper, 我们将该可执行文件作为MapReduce程序的Mapper。

步骤2 编写作业提交脚本。

编写应用程序提交脚本, 内容如下:

```
bin/hadoop jar contrib/streaming/hadoop-streaming-1.0.0.jar\
-mapper Mapper\
-reducer NONE\
-input/home/dongxicheng/input\
-output/home/dongxicheng/output\
-file Mapper\
-file debug_map.sh\
-mapdebug./debug_map.sh
```

其中, Map Task调试脚本debug_map.sh^[1]内容如下:

```
core=find.-name'core*';
gdb-quiet./Mapper-c$core-ex'info threads'-ex'backtrace'-ex'quit'
```

步骤3 准备好输入数据后, 提交作业, 可在\$Hadoop.log.dir/userlogs/\$jobid/\$taskid/debugout中看到调试脚本输出结果。

\$Hadoop.log.dir/userlogs/\$jobid/\$taskid/log.index: 日志索引文件。当JVM允许复用时, 所有复用同一个JVM的任务会将日志保存

在第一个任务的日志文件中，因此，需要一个日志索引文件保存日志实际存放位置以及每个任务对应的标准输出日志、标准错误输出日志和系统日志在文件`stdout`、`stderr`和`syslog`中的偏移量，具体格式如下：

```
LOG DIR: <the dir where the task logs are really stored>
stdout: <start-offset in the stdout file><length>
stderr: <start-offset in the stderr file><length>
syslog: <start-offset in the syslog file><length>
```

[1] 由于调试信息是从`core`文件中获取的，因此提交作业前应修改Linux相关配置以确保能够生成`core`文件。<http://lxc.sourceforge.net/>

文件`/proc/<pid>/stat`中包含的内存大小单位为`page`。为了获取以字节为单位的内存信息，`TaskTracker`通过执行以下`Shell`命令获取每个`page`对应的内存量（单位：B）：

```
getconf PAGESIZE
```

通过以上信息可计算当前每个运行的任务使用的内存总量。但需要注意的是，不能仅凭该内存量是否超过设定的内存最高值决定杀死一个任务。在创建一个子进程时，`JVM`采用了“`fork()+exec()`”模型，这意味着进程创建之后、执行之前会复制一份父进程内存空间，进而使得进程树在某一小段时间内存使用量翻倍。为了避免误杀任务，`Hadoop`赋予每个进程“年龄”属性，并规定刚启动进程的年龄是1，且`TaskMemoryManagerThread`线程每更新一次，各个进程年龄加1。在此基础上，选择被杀死任务的标准如下：

如果一个任务对应的进程树中所有进程（年龄大于0）总内存超过（用户设置的）最大值的两倍，或者所有年龄大于1的进程总内存量超过（用户设置的）最大值，则认为该任务过量使用内存，直接将其杀死，并将状态标注为`FAILED`。

步骤3 判断任务总内存使用量是否超过总可用内存量。

计算所有正在运行的任务当前使用的内存总量，如果超过系统可用内存总量（`slot`数与`slot`对应内存乘积），则`TaskTracker`会不断选择进度最慢的任务并将其杀掉，直到内存使用量降到可用内存总量以下。

需要注意的是，在下一代`MapReduce`（见第12章）中，同时增加了对内存资源和CPU资源的隔离机制，但采用的资源隔离方案不同。对于内存资源，为了能够更灵活地控制内存使用量，它仍采用了本节所述的线程监控的方案。采用这种机制的主要原因是`Java`中创建子进程采用了“`fork()+exec()`”的方案，子进程创建瞬间，它使用的内存量与父进程一致，从外面看来，一个进程使用内存量瞬间翻倍，然后又降下来，采用线程监控的方法可防止这种情况下导致`swap`操作。对于CPU资源，则采用了`Cgroups`进行资源隔离。具体可参考`YARN-3` [2]。

[1] 这里的内存指的是虚拟内存。从0.21.0版本开始，`Hadoop`允许设置物理内存，具体可参考：
<https://issues.apache.org/jira/browse/MAPREDUCE-1221>。

[2] <https://issues.apache.org/jira/browse/YARN-3>

7.7 小结

本章从TaskTracker架构、TaskTracker行为、作业目录管理等几个方面深入分析了TaskTracker工作原理及其实现。

TaskTracker以服务的形式存在，通过心跳机制向JobTracker汇报任务运行状态，并索取来自JobTracker的各种命令。

TaskTracker收到的命令包括启动任务（LaunchTaskAction）、提交任务（CommitTaskAction）、杀死任务（KillTaskAction）、杀死作业（KillJobAction）和重新初始化（TaskTrackerReinitAction）五种。TaskTracker收到这些命令后，会按照要求执行相应的操作。

TaskTracker最重要的功能之一是启动新任务。一个任务的启动过程大体包括两步：作业本地化和任务启动。为了进行资源隔离，TaskTracker为每个任务启动独立的Java虚拟机。此外，TaskTracker启动了一个额外的内存监控进程以防止任务滥用内存资源。

总体上说，TaskTracker扮演着“通信枢纽”的角色，是JobTracker与Task之间的“沟通桥梁”。

至此，我们已经剖析了JobTracker和TaskTracker两个重要服务的实现。接下来，我们将介绍Task实现，包括Map Task和Reduce Task两种Task的内部实现细节。

第8章 Task运行过程分析

大家都知道，当我们需要编写一个简单的MapReduce作业时，只需实现map()和reduce()两个函数即可，一旦将作业提交到集群上后，Hadoop内部会将这两个函数封装到Map Task和Reduce Task中，同时将它们调度到多个节点上并行执行，而任务执行过程中可能涉及的数据跨节点传输，记录按key分组等操作均由Task内部实现好了，用户无须关心。

为了帮助读者深入了解Map Task和Reduce Task内部实现原理，在本章中，我们将Map Task分解成Read、Map、Collect、Spill和Combine五个阶段，将Reduce Task分解成Shuffle、Merge、Sort、Reduce和Write五个阶段，并依次详细剖析每个阶段的内部实现细节。

8.1 Task运行过程概述

在MapReduce计算框架中，一个应用程序被划分成Map和Reduce两个计算阶段，它们分别由一个或者多个Map Task和Reduce Task组成。其中，每个Map Task处理输入数据集合中的一片数据（InputSplit），并将产生的若干个数据片段写到本地磁盘上，而Reduce Task则从每个Map Task上远程拷贝相应的数据片段，经分组聚集和归约后，将结果写到HDFS上作为最终结果，具体如图8-1所示。总体上看，Map Task与Reduce Task之间的数据传输采用了pull模型。为了能够容错，Map Task将中间计算结果存放到本地磁盘上，而Reduce Task则通过HTTP请求从各个Map Task端拖取（pull）相应的输入数据。为了更好地支持大量Reduce Task并发从Map Task端拷贝数据，Hadoop采用了Jetty Server作为HTTP Server处理并发数据读请求。

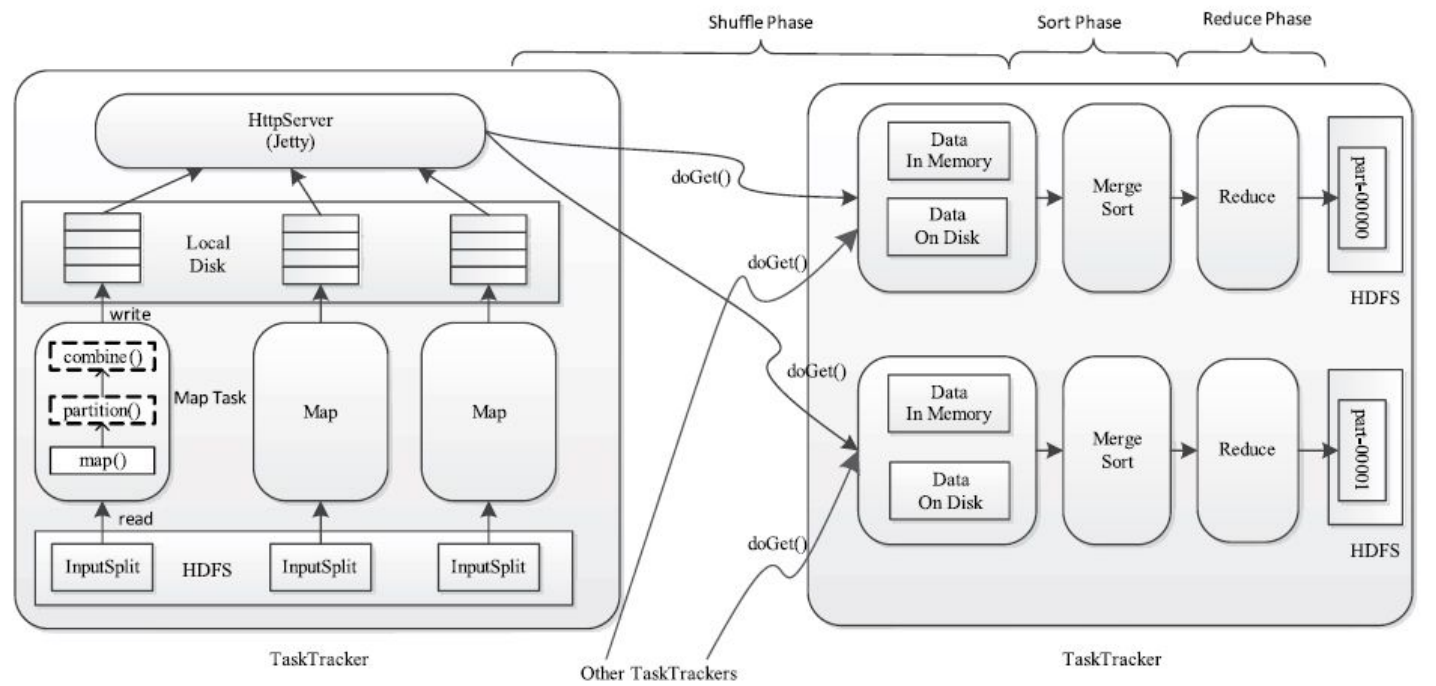


图 8-1 Map/Reduce Task运行过程

对于Map Task而言，它的执行过程可概述为：首先，通过用户提供的InputFormat将对应的InputSplit解析成一系列key/value，并依次交给用户编写的map()函数处理；接着按照指定的Partitioner对数据分片，以确定每个key/value将交给哪个Reduce Task处理；之后将数据交给用户定义的Combiner进行一次本地规约（用户没有定义则直接跳过）；最后将处理结果保存到本地磁盘上。

对于Reduce Task而言，由于它的输入数据来自各个Map Task，因此首先需通过HTTP请求从各个已经运行完成的Map Task上拷贝对应的数据分片，待所有数据拷贝完成后，再以key为关键字对所有数据进行排序，通过排序，key相同的记录聚集到一起形成若干分组，然后将每组数据交给用户编写的reduce()函数处理，并将数据结果直接写到HDFS上作为最终输出结果。

在接下来的几节中，我们将逐步深入分析Map Task和Reduce Task内部实现，并剖析其设计原理和实现细节。

8.2 基本数据结构和算法

在Map Task和Reduce Task实现过程中用到了大量数据结构和算法，我们选取了其中几个非常核心的部分进行介绍。

前面提到，用户可通过InputFormat和OutputFormat两个组件自定义作业的输入输出格式，但并不能自定义Map Task的输出格式（也就是Reduce Task的输入格式）。考虑到Map Task的输出文件需要到磁盘上并被Reduce Task远程拷贝，为尽可能减少数据量以避免不必要的磁盘和网络开销，Hadoop内部实现了支持行压缩的数据存储格式——IFile。

按照MapReduce语义，Reduce Task需将拷贝自各个Map Task端的数据按照key进行分组后才能交给reduce()函数处理，为此，Hadoop实现了基于排序的分组算法。但考虑到若完全由Reduce Task进行全局排序会产生性能瓶颈，Hadoop采用了分布式排序策略：先由各个Map Task对输出数据进行一次局部排序，然后由Reduce Task进行一次全局排序。

在任务运行过程中，为了能够让JobTracker获取任务执行进度，各个任务会创建一个进度汇报线程Reporter，只要任务处理一条新数据，该线程将通过RPC告知TaskTracker，并由TaskTracker通过心跳进一步告诉JobTracker。

8.2.1 IFile存储格式

IFile是一种支持行压缩的存储格式。通常而言，Map Task中间输出结果和Reduce Task远程拷贝结果被存放在IFile格式的磁盘文件或者内存文件中。为了尽可能减少Map Task写入磁盘数据量和跨网络传输数据量，IFile支持按行压缩数据记录。当前Hadoop提供了Zlib（默认压缩方式）、BZip2等压缩算法。如果用户想启用数据压缩功能，则需为作业添加以下两个配置选项。

❑ `mapred.compress.map.output`：是否支持中间输出结果压缩，默认为false。

❑ `mapred.map.output.compression.codec`：压缩器（默认是基于Zlib算法的压缩器DefaultCodec）。任何一个压缩器需实现CompressionCodec接口以提供压缩输出流和解压缩输入流。

一旦启用了压缩机制，Hadoop会为每条记录的key和value值进行压缩。

IFile定义的文件格式非常简单，整个文件顺次保存数据记录，每条数据记录格式为：

```
<key-len, value-len, key, value>
```

由于Map Task会按照key值对输出数据进行排序，因此IFile通常保存的是有序数据集。

IFile文件读写操作由类IFile实现，该类中包含两个重要内部类：Writer和Reader，分别用于Map Task生成IFile和Reduce Task读取一个IFile（对于内存中的数据读取，则使用InMemoryReader）。此外，为了保证数据一致性，Hadoop分别为Writer和Reader提供了IFileOutputStream和IFileInputStream两个支持CRC32校验的类，具体如图8-2所示。

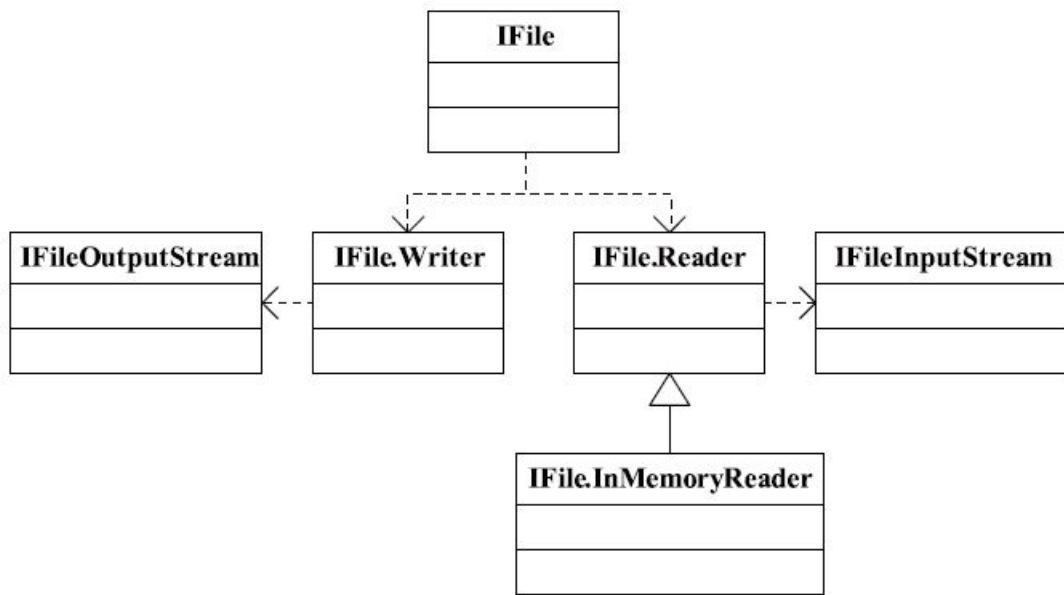


图 8-2 IFile类关系图

8.2.2 排序

排序是MapReduce框架中最重要的操作之一。Map Task和Reduce Task均会对数据（按照key）进行排序。该操作属于Hadoop的默认行为。任何应用程序中的数据均会被排序，而不管逻辑上是否需要。

对于Map Task，它会将处理的结果暂时放到一个缓冲区中，当缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次排序，并将这些有序数据以IFile文件形式写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行一次合并，以将这些文件合并成一个大的有序文件。

对于Reduce Task，它从每个Map Task上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则放到磁盘上，否则放到内存中。如果磁盘上文件数目达到一定阈值，则进行一次合并以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据写到磁盘上。当所有数据拷贝完毕后，Reduce Task统一对内存和磁盘上的所有数据进行一次合并。

在Map Task和Reduce Task运行过程中，缓冲区数据排序使用了Hadoop自己实现快速排序算法，而IFile文件合并则使用了基于堆实现的优先队列。

1.快速排序

快速排序是应用最广泛的排序算法之一。它的基本思想是，选择序列中的一个元素作为枢轴，将小于枢轴的元素放在左边，将大于枢轴的元素放在右边，针对左右两个子序列重复此过程，直到序列为空或者只剩下一个元素。

在《算法导论》^[1]一书中，给出了一个教科书式的快速排序实现算法，它的实现方法是：选择序列的最后一个元素作为枢轴，并使用一个索引由前往后遍历整个序列，将小于枢轴的元素交换到左边，大于枢轴的元素交换到右边，直到序列为空或者只剩下一个元素。

Hadoop实现的快速排序在该快速排序之上进行了以下优化。

（1）枢轴选择

枢轴的选择好坏直接影响快速排序的性能，而最坏的情况是划分过程中始终产生两个极端不对称的子序列（有一个长度为1，另一个为 $n-1$ ），此时排序算法复杂度将增为 $O(N^2)$ 。减小出现划分严重不对称的可能性，Hadoop将序列的首尾和中间元素中的中位数作为枢轴。

（2）子序列划分方法

Hadoop使用了两个索引i和j分别从左右两端进行扫描序列，并让索引i扫描到大于等于枢轴的元素停止，索引j扫描到小于等于枢轴的元素停止，然后交换两个元素，重复这个过程直到两个索引相遇。

（3）对相同元素的优化

在每次划分子序列时，将与枢轴相同的元素集中存放到中间位置，让它们不再参与后续的递归处理，即将序列划分成三部分：小于枢轴、等于枢轴和大于枢轴。

（4）减少递归次数

当子序列中元素数目小于13时，直接使用插入排序算法，不再继续递归。

2.优先队列

文件归并由类Merger完成，它要求待排序对象需是Segment实例化对象。Segment是对磁盘和内存中的IFile格式文件的抽象。它具有类似于迭代器的功能，可迭代读取IFile文件中的key/value。

Merger采用了多轮递归合并的方式，每轮选取最小的前io.sort.factor（默认是10，用户可配置）个文件进行合并，并将产生的文件重新加入待合并列表中，直到剩下的文件数目小于io.sort.factor个，此时，它会返回指向由这些文件组成的小顶堆的迭代器。在图8-3中，我们给出了一个io.sort.factor为3的文件合并实例。

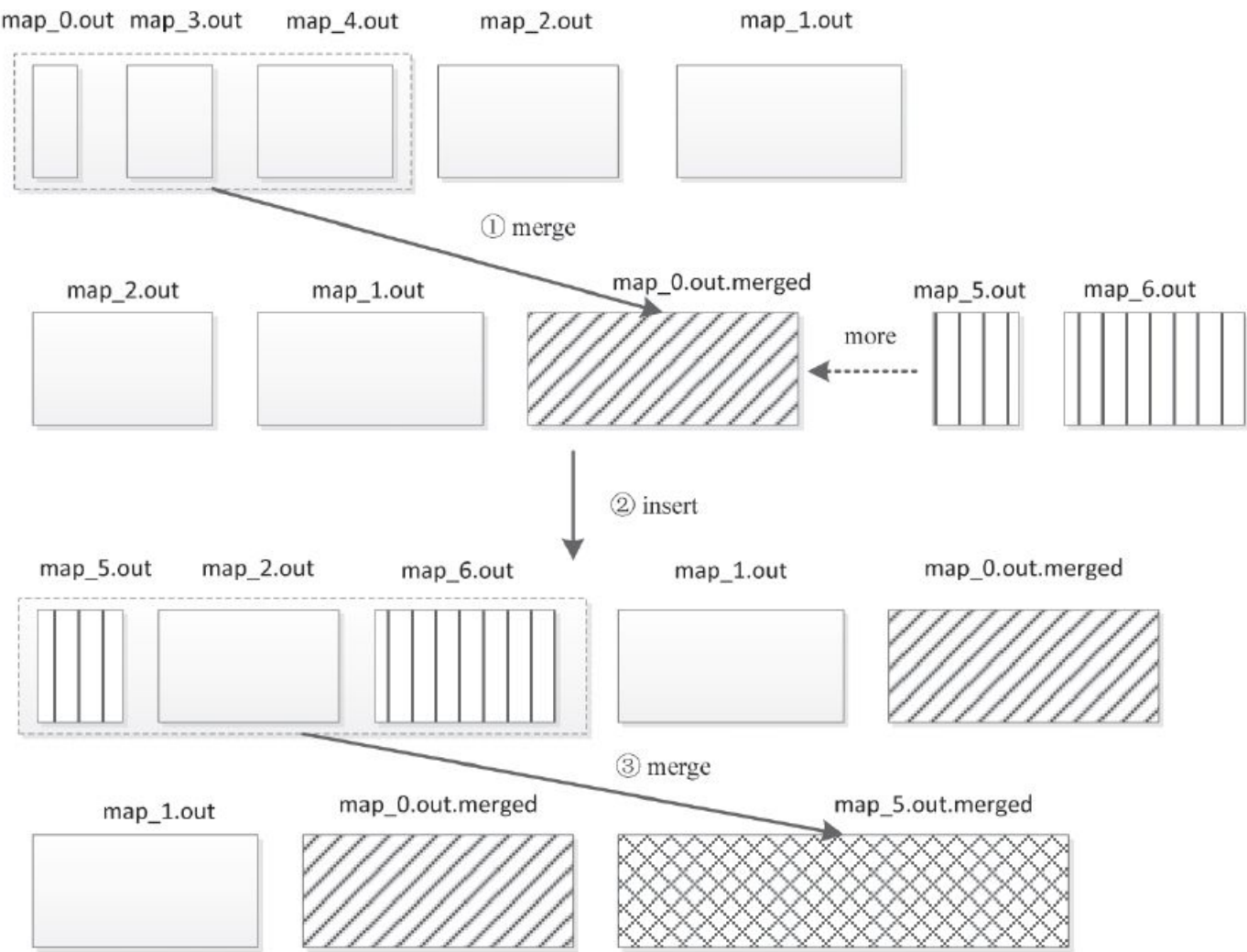


图 8-3 文件合并过程

如图8-4所示，在每一轮合并过程中，Merger采用了小顶堆实现，进而可将文件合并过程看作一个不断建堆的过程：建堆→取堆顶元素→重新建堆→取堆顶元素……

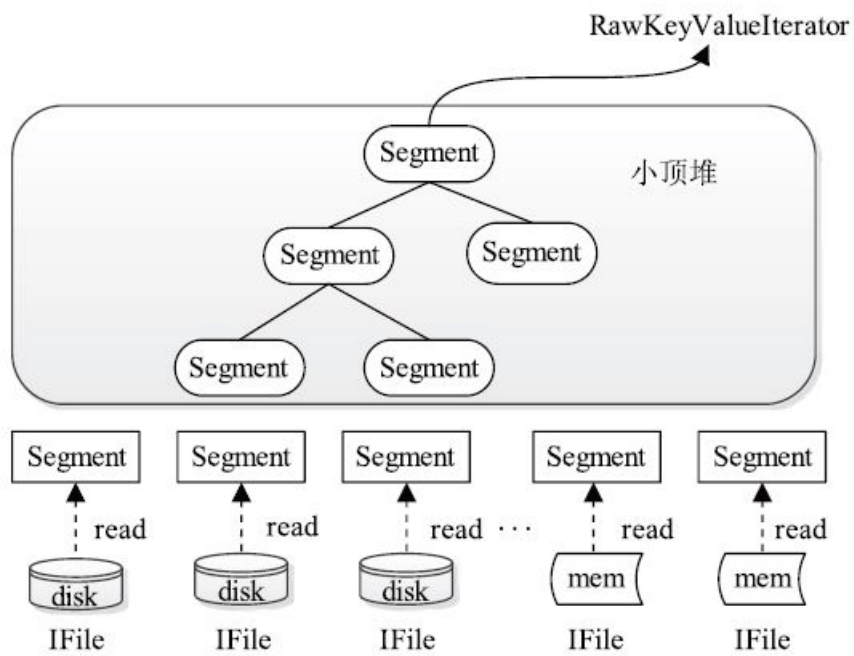


图 8-4 利用小顶堆合并文件

[1] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest、Clifford Stein, “算法导论”, 第3版。

8.2.3 Reporter

前几章提到，所有Task需周期性向TaskTracker汇报最新进度和计数器值，而这正是由Reporter组件实现的。在Map/Reduce Task中，TaskReporter类实现了Reporter接口，并且以线程形式启动。TaskReporter汇报的信息中包含两部分：任务执行进度和任务计数器值。

1.任务执行进度

任务执行进度信息被封装到类Progress中，且每个Progress实例以树的形式存在。Hadoop采用了简单的线性模型计算每个阶段的进度值：如果一个大阶段可被分解成若干个子阶段，则可将大阶段看作一棵树的父节点，而子阶段可看作父节点对应的子节点，且大阶段的进度值可被均摊到各个子阶段中；如果一个阶段不可再分解，则该阶段进度值可表示成已读取数据量占总数据量的比例。

对于Map Task而言，它作为一个大阶段不可再分解，为了简便，我们直接将已读取数据量占总数据量的比例作为任务当前执行进度值。

对于Reduce Task而言，我们可将其分解成三个阶段：Shuffle、Sort和Reduce，每个阶段占任务总进度的1/3。考虑到在Shuffle阶段，Reduce Task需从M（M为Map Task数目）个Map Task上读取一片数据，因此，可被分解成M个阶段，每个阶段占Shuffle进度的1/M，具体如图8-5所示。

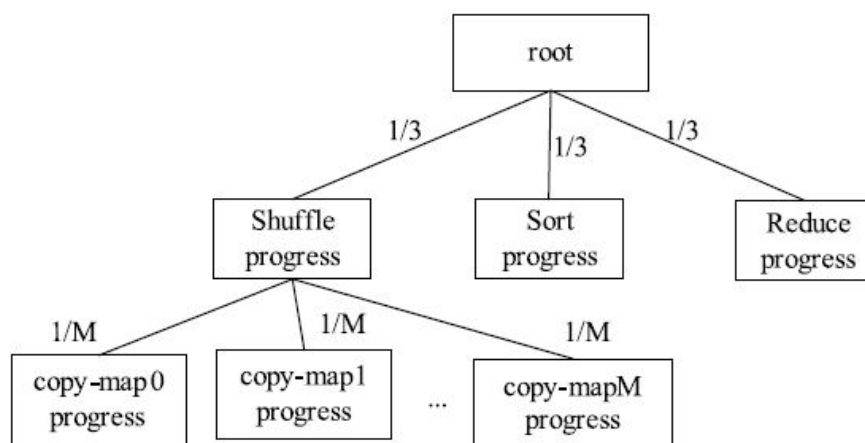


图 8-5 Reduce Task进度树

对于TaskReporter线程而言，它并不会总是每隔一段时间汇报进度和计数器值，而是仅当发现以下两种情况之一时才会汇报。

- ❑ 任务执行进度发生变化；
- ❑ 任务的某个计数器值发生变化。

在某个时间间隔内，如果任务执行进度和计数器值均未发生变化，则Task只会简单地通过调用RPC函数ping探测TaskTracker是否活着。在一定时间内，如果某个任务的执行进度和计数器值均未发生变化，则TaskTracker认为它处于悬挂（hang up）状态，直接将其杀掉。为了防止某条记录因处理时间过长导致被杀，用户可采用以下两种方法：

- ❑ 每隔一段时间调用一次TaskReporter.progress()函数，以告诉TaskTracker自己仍然活着。
- ❑ 增大任务超时参数mapred.task.timeout（默认是10 min）对应的值。

2.任务计数器

任务计数器（Counter）是Hadoop提供的，用于实现跟踪任务运行进度的全局计数功能。用户可在自己的应用程序中添加计数器。任务计数器由两部分组成：<name, value>，其中，name表示计数器名称，value表示计数器值（long类型）。计数器通常以组为单位管理，一个计数器属于一个计数器组（CounterGroup）。此外，Hadoop规定一个作业最多包含120个计数器（可通过参数mapreduce.job.counters.limit设定），50个计数器组。

对于同一个任务而言，所有任务包含的计数器相同，每个任务更新自己的计数器值，然后汇报给TaskTracker，并由TaskTracker通过心跳汇报给JobTracker，最后由JobTracker以作业为单位对所有计数器进行累加。作业的计数器分为两类：MapReduce内置计数器和用户自定义计数器。

(1) MapReduce内置计数器

MapReduce框架内部为每个任务添加了三个计数器组，分别位于File Input Format Counters, File Output Format Counters和Map-Reduce Framework中。它们包含的计数器分别见表8-1，表8-2和表8-3。

表 8-1 File Input Format Counters 中的计数器

计数器名称	计数器含义	计数器更新
Bytes Read	从文件系统中读入数据量（B）	TrackedRecordReader

表 8-2 File Output Format Counters 中的计数器

计数器名称	计数器含义	计数器更新
Bytes Written	写入文件系统的数据量（B）	OldTrackingRecordWriter/ NewTrackingRecordWriter

表 8-3 Map-Reduce Framework 中的计数器

计数器名称	计数器含义	计数器更新
Map input bytes	Map Task 输入数据量（B）	TrackedRecordReader
Map output records	Map Task 输入记录条数	MapOutputBuffer
Map output bytes	Map Task 输出数据量（B）	MapOutputBuffer
Map output materialized bytes	Map Task 输出数据量（B）	MapOutputBuffer
Map skipped records	Map Task 跳过坏记录条数	SkippingRecordReader
Combine input records	Combiner 输入记录条数	MapOutputBuffer
Combine output records	Combiner 输出记录条数	MapOutputBuffer
Reduce input groups	Reduce Task 输入分组数目	runOldReducer/runNewReducer

(续)

计数器名称	计数器含义	计数器更新
Reduce shuffle bytes	Shuffle 数据量 (B)	MapOutputCopier
Reduce input records	Reduce Task 输入记录条数	runOldReducer/runNewReducer
Reduce output records	Reduce Task 输出记录条数	OldTrackingRecordWriter/ NewTrackingRecordWriter
Reduce skipped records	Reduce Task 跳过坏记录条数	SkippingReduceValuesIterator
Reduce skipped groups	Reduce Task 跳过分组数目	SkippingReduceValuesIterator
Spilled Records	溢写记录条数	MapOutputBuffer
Total committed heap usage (bytes)	堆栈使用量 (B)	TaskReporter
CPU time spent (ms)	耗费 CPU 时间	TaskReporter
Physical memory (bytes) snapshot	耗费物理内存量	TaskReporter
Virtual memory (bytes) snapshot	耗费虚拟内存量	TaskReporter

(2) 用户自定义计数器

不同的编程接口，定义计数器的方式不同。接下来，我们简要介绍Java、Hadoop Pipes和Hadoop Streaming中定义计数器的方法。

Java

Hadoop为Java应用程序提供了两种访问和使用计数器的方式：使用枚举类型和字符串类型。如果采用枚举类型，则计数器默认名称是枚举类型的Java完全限定类名，这使得计数器名称的可读性很差，为此，Hadoop提供了基于资源捆绑修改计数器显示名称的方法：以Java枚举类型为名称创建一个属性文件。在该属性文件中，“CounterGroupName”属性用于设定整个组的显示名称，而枚举类型中每个字段均有一个属性与之对应，属性名称为“字段类型.name”，属性值即为该计数器的显示名称。比如，类Task中定义了大量表示计数器的枚举类型，而这些计数器的显示名称被统一放到同目录下的属性文件Task_Counter.properties中，且内容如下：

```
CounterGroupName=
Map-Reduce Framework
MAP_INPUT_RECORDS.name=
Map_input_records
MAP_INPUT_BYTES.name=
Map_input_bytes
MAP_OUTPUT_RECORDS.name=
Map_output_records
MAP_OUTPUT_BYTES.name=
Map_output_bytes
.....
```

如果采用字符串类型，则用户可以直接在计数器API中指定计数器组，计数器名称和计数器值。基于枚举类型和字符串类型的计数器API如下：

```
public abstract void incrCounter (Enum<?>key, long amount);
public abstract void incrCounter (String group, String counter, long amount);
```

Hadoop Pipes

Hadoop Pipes提供了一套基于字符串类型的计数器API。通常使用一个计数器需分成三步，分别是定义、注册和使用，举例如下：

```
HadoopPipes: TaskContext: Counter*mapCounter; //定义
mapCounter=context.getCounter ("counterGroup", "mapCounter"); //注册
```



```
context.incrementCounter (mapCounter, 1); //使用
```

Hadoop Streaming

前面提到，**Hadoop Streaming**基于标准输入输出机制可支持多种语言编写**MapReduce**程序，而标准输入输出流包含三种：标准输入流、标准输出流和标准错误输出流，其中前两个主要用于输入输出数据，第三种则用于向**Java**端传递任务运行状态，包括计数器值、任务状态等。如果用户发送计数器值，则可使用标准错误输出流输出以下字符串：

```
reporter: counter: <group>, <counter>, <amount>
```

比如，使用**C**语言可使用以下程序段增加计数器值：

```
cerr<<"reporter: counter: counterGroup, mapCounter, 1"
```

8.3 Map Task内部实现

前面提到，Map Task分为4种，分别是Job-setup Task、Job-cleanup Task、Task-cleanup Task和Map Task。其中，Job-setup Task和Job-cleanup Task分别是作业运行时启动的第一个任务和最后一个任务，主要工作分别是进行一些作业初始化和收尾工作，比如创建和删除作业临时输出目录；而Task-cleanup Task则是任务失败或者被杀死后，用于清理已写入临时目录中数据的任务。本节主要讲解第四种任务——普通的Map Task。它需要处理数据，并将计算结果存到本地磁盘上。

8.3.1 Map Task整体流程

- Map Task的整体计算流程如图8-6所示，共分为5个阶段，分别是：
- Read阶段：Map Task通过用户编写的RecordReader，从输入InputSplit中解析出一个个key/value。
 - Map阶段：该阶段主要是将解析出的key/value交给用户编写的map()函数处理，并产生一系列新的key/value。
 - Collect阶段：在用户编写的map()函数中，当数据处理完成后，一般会调用OutputCollector.collect()输出结果。在该函数内部，它会将生成的key/value分片（通过调用Partitioner），并写入一个环形内存缓冲区中。
 - Spill阶段：即“溢写”，当环形缓冲区满后，MapReduce会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次本地排序，并在必要时对数据进行合并、压缩等操作。
 - Combine阶段：当所有数据处理完成后，Map Task对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

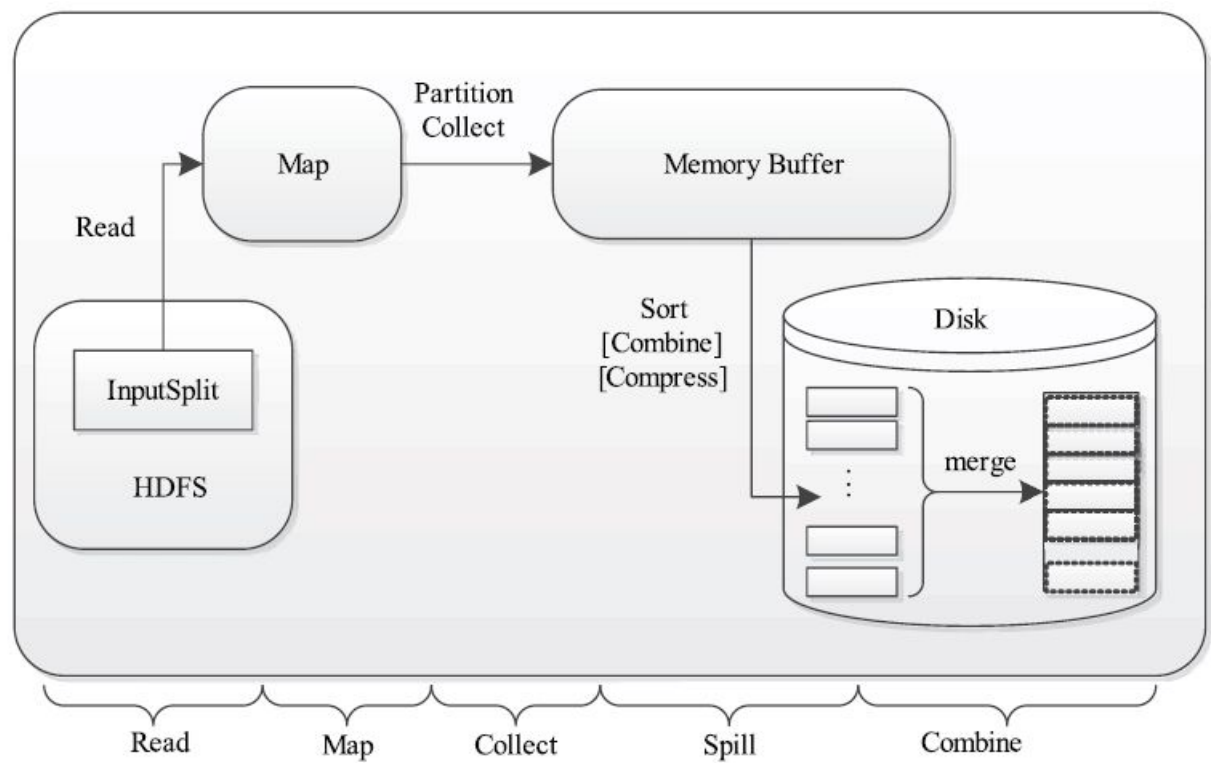


图 8-6 Map Task计算流程

MapReduce框架提供了两套API，默认情况下采用旧API，用户可通过设置参数mapred.mapper.new-api为true启用新API。新API在封装性和扩展性等方面优于旧API，但性能上并没有改进。本章主要以旧API为例进行讲解。

在Map Task中，最重要的部分是输出结果在内存和磁盘中的组织方式，具体涉及Collect、Spill和Combine三个阶段，也就是用

户调用`OutputCollector.collect()`函数之后依次经历的几个阶段。我们将在下面几小节深入分析这几个阶段。

8.3.2 Collect过程分析

待map()函数处理完一对key/value，并产生新的key/value后，会调用OutputCollector.collect()函数输出结果。本小节重点剖析该函数内部实现机制。

跟踪进入Map Task的入口函数run()，可发现，如果用户选用旧API，则会调用runOldMapper函数处理数据。该函数根据实际的配置创建合适的MapRunnable以迭代调用用户编写的map()函数，而map()函数的参数OutputCollector正是MapRunnable传入的OldOutputCollector对象。

OldOutputCollector根据作业是否包含Reduce Task封装了不同的MapOutputCollector实现，如果Reduce Task数目为0，则封装DirectMapOutputCollector对象直接将结果写入HDFS中作为最终结果，否则封装MapOutputBuffer对象暂时将结果写入本地磁盘上以供Reduce Task进一步处理。本小节主要分析Reduce Task数目非0的情况。

用户在map()函数中调用OldOutputCollector.collect(key, value)后，在该函数内部，首先会调用Partitioner.getPartition()函数获取记录的分区号partition，然后将三元组<key, value, partition>传递给MapOutputBuffer.collect()函数做进一步处理。

MapOutputBuffer内部使用了一个缓冲区暂时存储用户输出数据，当缓冲区使用率达到一定阈值后，再将缓冲区中的数据写到磁盘上。数据缓冲区的设计方式直接影响到Map Task的写效率，而现有多种数据结构可供选择，最简单的是单向缓冲区，生产者向缓冲区中单向写入输出，当缓冲区写满后，一次性写到磁盘上，就这样，不断写缓冲区，直到所有数据写到磁盘上。单向缓冲区最大的问题是性能不高，不能支持同时读写数据。双缓冲区是对单向缓冲区的一个改进，它使用两个缓冲区，其中一个用于写入数据，另一个将写满的数据写到磁盘上，这样，两个缓冲区交替读写，进而提高效率。实际上，双缓冲区只能一定程度上让读写并行，仍会存在读写等待问题。一种更好的缓冲区设计方式是采用环形缓冲区：当缓冲区使用率达到一定阈值后，便开始向磁盘上写入数据，同时，生产者仍可以向不断增加的剩余空间中循环写入数据，进而达到真正的读写并行。三种缓冲区结构如图8-7所示。

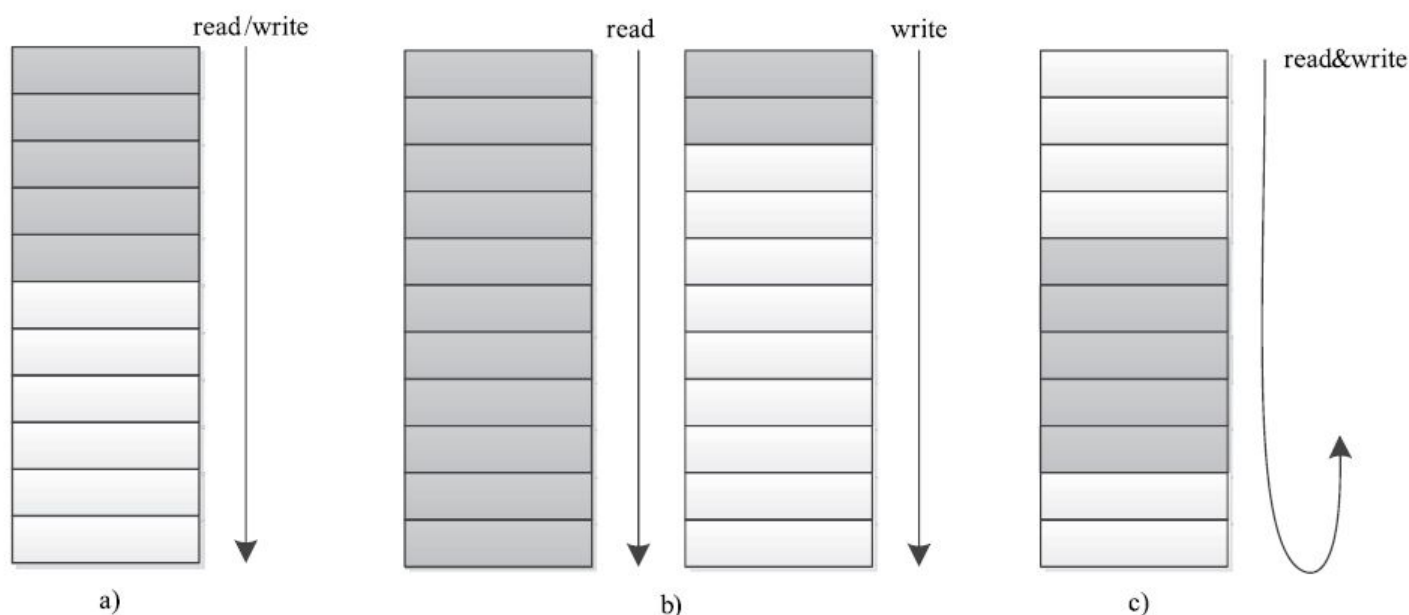


图 8-7 三种缓冲区结构

a) 单向缓冲区 b) 双缓冲区 c) 循环缓冲区

MapOutputBuffer正是采用了环形内存缓冲区保存数据^[1]，当缓冲区使用率达到一定阈值后，由线程SpillThread将数据写到一个临时文件中，当所有数据处理完毕后，对所有临时文件进行一次合并以生成一个最终文件。环形缓冲区使得Map Task的Collect阶段和Spill阶段可并行进行。

MapOutputBuffer内部采用了两级索引结构（见图8-8），涉及三个环形内存缓冲区，分别是kvooffsets、kvindices和kvbuffer，这三个缓冲区所占内存空间总大小为io.sort.mb（默认是100 MB）。下面分别介绍这三个缓冲区的含义。

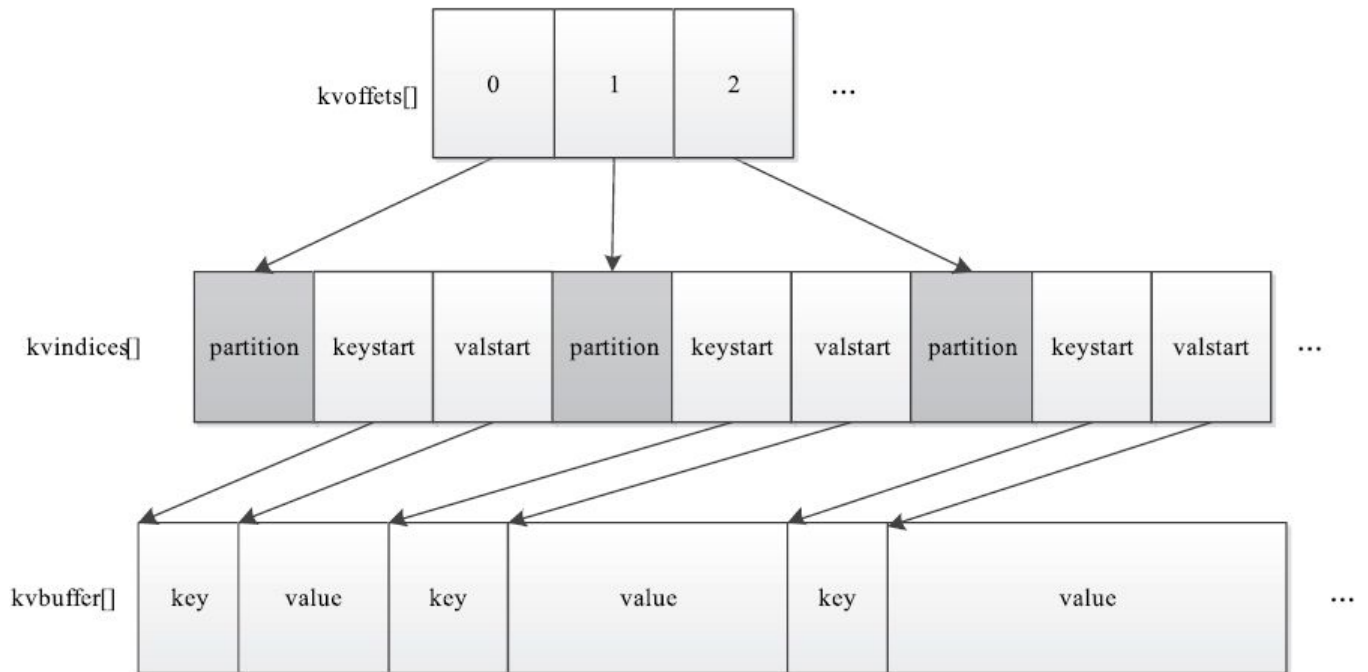


图 8-8 MapOutputBuffer的两级索引结构

(1) kvoffsets

kvoffsets即偏移量索引数组，用于保存key/value信息在位置索引kvindices中的偏移量。考虑到一对key/value需占用数组kvoffsets的1个int（整型）大小，数组kvindices的3个int大小（分别保存所在partition号、key开始位置和value开始位置），所以Hadoop按比例1: 3将大小为 $\text{\$}\{\text{io.sort.record.percent}\} * \text{\$}\{\text{io.sort.mb}\}$ 的内存空间分配给数组kvoffsets和kvindices，其间涉及的缓冲区分配方式见图8-9，计算过程如下：

```
private static final int ACCTSIZE=3; //每对key/value占用kvindices中的三项
private static final int RECSIZE=(ACCTSIZE+1) *4; //每对key/value共占用
kvoffsets和kvindices中的4个字节 (4*4=16 byte)
final float recper=job.getFloat("io.sort.record.percent", (float) 0.05);
final int sortmb=job.getInt("io.sort.mb", 100);
int maxMemUsage=sortmb<<20; //将内存单位转化为字节
int recordCapacity=(int) (maxMemUsage*recper);
recordCapacity-=recordCapacity%RECSIZE; //保证recordCapacity是4*4的整数倍
recordCapacity/=RECSIZE; //计算内存中最多保存key/value数目
kvoffsets=new int[recordCapacity]; //kvoffsets占用1: 3中的1
kvindices=new int[recordCapacity*ACCTSIZE]; //kvindices占用1: 3中的3
```

当该数组使用率超过io.sort.spill.percent后，便会触发线程SpillThread将数据写入磁盘。

(2) kvindices

kvindices即位置索引数组，用于保存key/value值在数据缓冲区kvbuffer中的起始位置。

(3) kvbuffer

kvbuffer即数据缓冲区，用于保存实际的key/value值，默认情况下最多可使用io.sort.mb中的95%，当该缓冲区使用率超过io.sort.spill.percent后，便会触发线程SpillThread将数据写入磁盘。

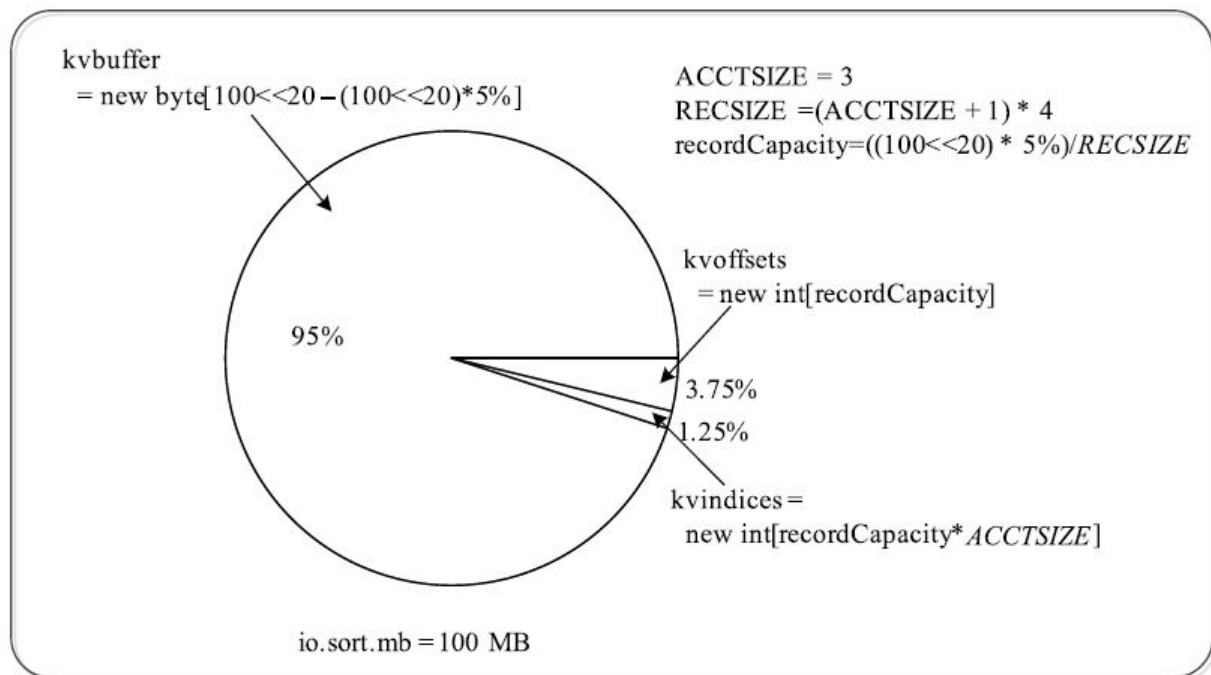


图 8-9 内存io.sort.mb的分配方式

以上几个缓冲区读写采用了典型的单生产者消费者模型，其中，MapOutputBuffer的collect方法和MapOutputBuffer.Buffer的write方法是生产者，spillThread线程是消费者，它们之间同步是通过可重入的互斥锁spillLock和spillLock上的两个条件变量（spillDone和spillReady）完成的。生产者主要的伪代码如下：

```
//取得下一个可写入的位置
spillLock.lock();
if (缓冲区使用率达到阈值) {
    //唤醒SpillThread线程，将缓冲区数据写入磁盘
    spillReady.signal();
}
if (缓冲区满) {
    //等待SpillThread线程结束
    spillDone.wait();
}
spillLock.lock();
//将数据写入缓冲区
```

下面分别介绍环形缓冲区kvoffsets和kvbuffer的数据写入过程。

(1) 环形缓冲区kvoffsets

通常用一个线性缓冲区模拟实现环形缓冲区，并通过取模操作实现循环数据存储。下面介绍环形缓冲区kvoffsets的写数据过程。该过程由指针kvstart/kvend/kvindex控制，其中kvstart表示存有数据的内存段初始位置，kvindex表示未存储数据的内存段初始位置，而在正常写入情况下，kvend=kvstart，一旦满足溢写条件，则kvend=kvindex，此时指针区间[kvstart, kvend)为有效数据区间。具体涉及的操作如下。

操作1：写入缓冲区。

直接将数据写入kvindex指针指向的内存空间，同时移动kvindex指向下一个可写入的内存空间首地址，kvindex移动公式为：
kvindex=(kvindex+1)%kvoffsets.length。由于kvoffsets为环形缓冲区，因此可能涉及两种写入情况。

情况1：kvindex>kvend，如图8-10所示。在这种情况下，指针kvindex在指针kvend后面，如果向缓冲区中写入一个字符串，则kvindex指针后移一位。

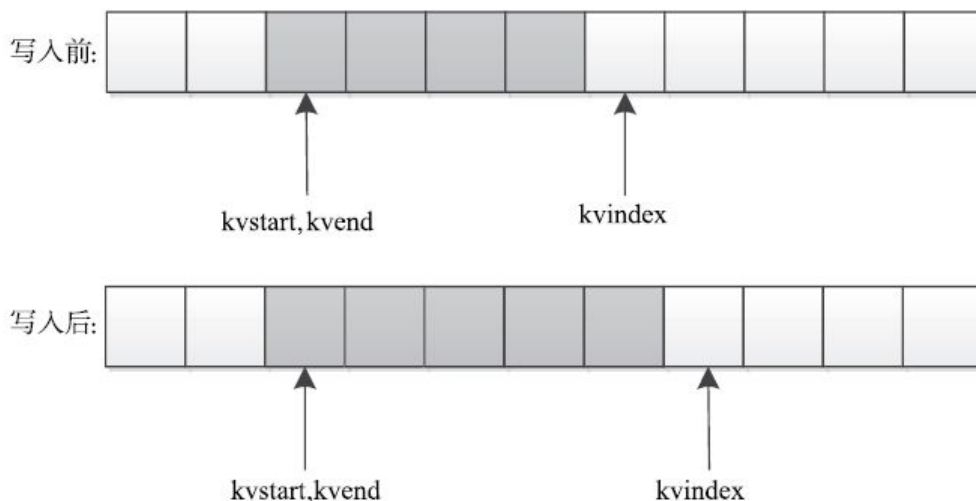


图 8-10 环形缓冲区在 $kvindex > kvend$ 情况下写入数据

情况2: $kvindex \leq kvend$, 如图8-11所示。在这种情况下, 指针 $kvindex$ 位于指针 $kvend$ 前面, 如果向缓冲区中写入一个字符串, 则 $kvindex$ 指针后移一位。

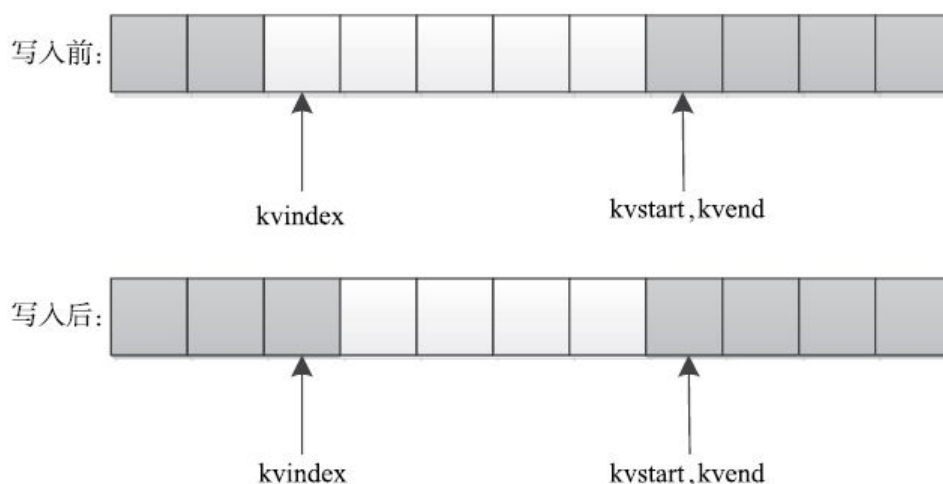


图 8-11 环形缓冲区在 $kvindex \leq kvend$ 情况下写入数据

操作2: 溢写到磁盘。

当 $kvoffsets$ 内存空间使用率超过 `io.sort.spill.percent` (默认是80%) 后, 需将内存中数据写到磁盘上。为了判断是否满足该条件, 需先求出 $kvoffsets$ 已使用内存。如果 $kvindex > kvend$, 则已使用内存大小为 $kvindex - kvend$; 否则, 已使用内存大小为 $kvoffsets.length - (kvend - kvindex)$ 。

(2) 环形缓冲区 `kvbuffer`

环形缓冲区 `kvbuffer` 的读写操作过程由指针 `bufstart/bufend/bufvoid/bufindex/bufmark` 控制, 其中, `bufstart/bufend/bufindex` 含义与 `kvstart/kvend/kvindex` 相同, 而 `bufvoid` 指向 `kvbuffer` 中有效内存结束为止, `kvbuffer` 表示最后写入的一个完整 `key/value` 结束位置, 具体写入过程中涉及的状态和操作如下:

情况1: 初始状态。

初始状态下, `bufstart=bufend=bufindex=bufmark=0`, `bufvoid=kvbuffer.length`, 如图8-12所示。



图 8-12 初始状态下的kvbuffer

情况2：写入一个key。

写入一个key后，需移动bufindex指针到可写入内存初始位置，如图8-13所示。



图 8-13 向kvbuffer中写入一个key

情况3：写入一个value。

写入key对应的value后，除移动bufindex指针外，还要移动bufmark指针，表示已经写入一个完整的key/value，具体如图8-14所示。



图 8-14 向kvbuffer中写入一个value

情况4：不断写入key/value，直到满足溢写条件，即kvoffsets或者kvbuffer空间使用率超过io.sort.spill.percent（默认值为80%）。此时需要将数据写到磁盘上，如图8-15所示。

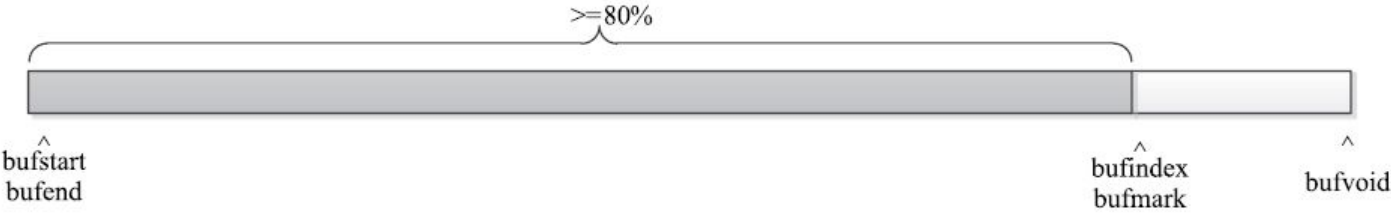


图 8-15 kvbuffer使用率达到阈值

情况5：溢写。

如果达到溢写条件，则令bufend←bufindex，并将缓冲区[bufstart, bufend)之间的数据写到磁盘上，具体如图8-16所示。

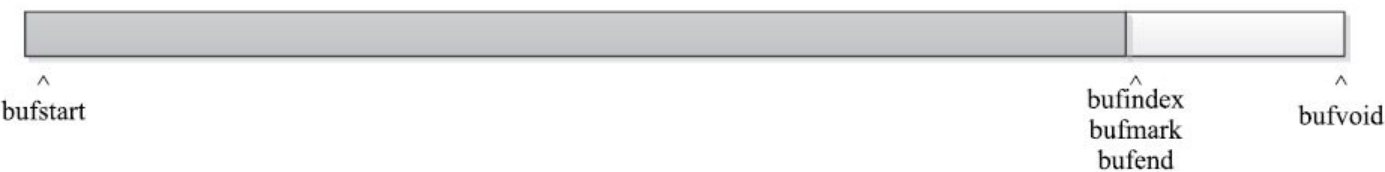


图 8-16 准备开始溢写

溢写完成之后，恢复正常写入状态，令bufstart←bufend，如图8-17所示。



图 8-17 溢写完成

在溢写的时候，Map Task仍可向kvbuffer中写入数据，如图8-18所示。

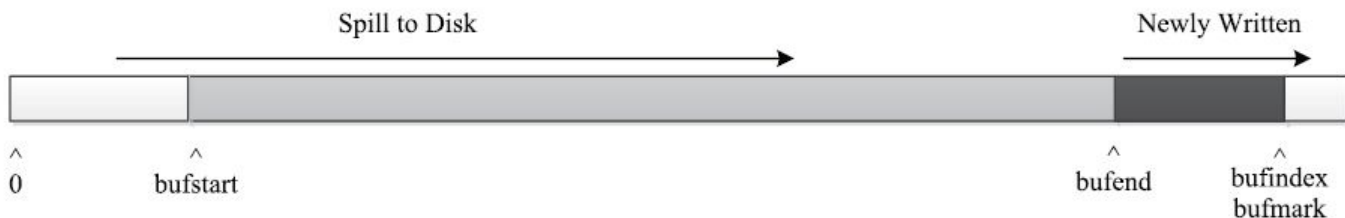


图 8-18 溢写的时候，被写入新数据

情况6：写入key时，发生跨界现象。

当写入某个key时，缓冲区尾部剩余空间不足以容纳整个key值，此时需要将key值分开存储，其中一部分存到缓冲区末尾，另外一部分存到缓冲区首部，具体如图8-19所示。



图 8-19 写入key时，发生越界

情况7：调整key位置，防止key跨界现象。

由于key是排序的关键字，通常需交给RawComparator进行排序，而它要求排序关键字必须在内存中连续存储，因此不允许key跨界存储。为解决该问题，Hadoop将跨界的key值重新存储到缓冲区的首位置，通常可分为以下两种情况。

□ $\text{bufindex} + (\text{bufvoid} - \text{bufmark}) < \text{bufstart}$: 此时缓冲区前半段有足够的空间容纳整个key值，因此可通过两次内存复制解决跨行问题，具体如图8-20所示。

```
int headbytelen=bufvoid-bufmark;
System.arraycopy(kvbuffer, 0, kvbuffer, headbytelen, bufindex);
System.arraycopy(kvbuffer, bufvoid, kvbuffer, 0, headbytelen);
```

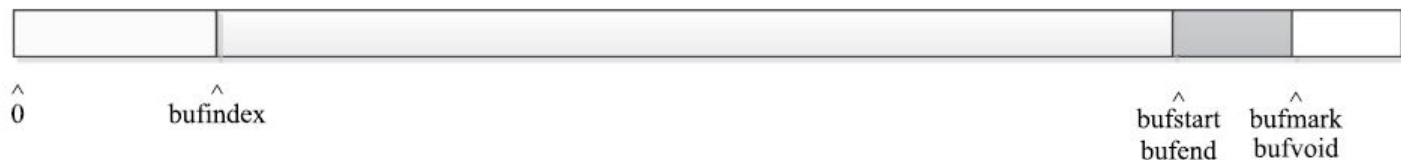


图 8-20 调整位置，避免key跨界

□ $\text{bufindex} + (\text{bufvoid} - \text{bufmark}) \geq \text{bufstart}$: 此时缓冲区前半段没有足够的空间容纳整个key值，将key值移到缓冲区开始位置时将触发一次Spill操作。这种情况下，可通过三次内存复制解决跨行问题：

```
byte[] keytmp=new byte[bufindex]; //申请一个临时缓冲区
System.arraycopy(kvbuffer, 0, keytmp, 0, bufindex);
bufindex=0;
out.write(kvbuffer, bufmark, headbytelen); //将key值写入缓冲区开始位置
```

```
out.write(keytmp);
```

情况8: 某个key或者value太大, 以至于整个缓冲区不能容纳它。

如果一条记录的key或value太大, 整个缓冲区都不能容纳它, 则Map Task会抛出MapBufferTooSmallException异常, 并将该记录单独输出到一个文件中。

(3) 环形缓冲区优化

在Hadoop 1.X版本中, 当满足以下两个条件之一时, Map Task会发生溢写现象。

条件1: 缓冲区kvindices或者kvbuffer的空间使用率达到io.sort.spill.percent (默认值为80%)。

条件2: 出现一条kvbuffer无法容纳的超大记录。

前面提到, Map Task将可用的缓冲区空间io.sort.mb按照一定比例 (由参数io.sort.record.percent决定) 静态分配给了kvooffsets、kvindices和kvbuffer三个缓冲区, 而正如条件1所述, 只要任何一个缓冲区的使用率达到一定比例, 就会发生溢写现象, 即使另外的缓冲区使用率非常低。因此, 设置合理的io.sort.record.percent参数, 对于充分利用缓冲区空间和减少溢写次数, 是十分必要的。考虑到每条数据 (一个key/value对) 需占用索引大小为16 B, 因此, 建议用户采用以下公式^[2]设置io.sort.record.percent:

$$\text{io.sort.record.percent} = 16 / (16 + R)$$

其中R为平均每条记录的长度。

【实例】假设一个作业的Map Task输入数据量和输出数据量相同, 每个Map Task输入数据量大小为128 MB, 且共有1 342 177条记录, 每条记录大小约为100 B, 则需要索引大小为 $16 * 1\,342\,177 = 20.9$ MB。根据这些信息, 可设置参数如下:

□ io.sort.mb: $128 \text{ MB} + 20.9 \text{ MB} = 148.9 \text{ MB}$

□ io.sort.record.percent: $16 / (16 + 100) = 0.138$

□ io.sort.spill.percent: 1.0

这样配置可保证数据只“落”一次地, 效率最高! 当然, 实际使用时可能很难达到这种情况, 比如每个Map Task输出数据量非常大, 缓冲区难以全部容纳它们, 但你至少可以设置合理的io.sort.record.percent以更充分地利用io.sort.mb并尽可能减少中间文件数目。

尽管用户可通过该经验公式设置一个较优的io.sort.record.percent参数, 但在实际应用中, 估算一个非常合理的R值仍是较麻烦的。为了从根本上解决这个问题, Hadoop 0.21采用共享环形缓冲区对Map Task输出数据的组织方式进行了优化, 这样, 用户无须再为自己的作业设置io.sort.record.percent参数。如图8-21所示, Hadoop 0.21主要有两个修改点^[3]:

□ 不再将索引和记录分放到不同的环形缓冲区中, 而是让它们共用一个环形缓冲区。

□ 引入一个新的指针equator。该指针界定了索引和数据共同起始存放位置。从该位置开始, 索引和数据分别沿相反的方向增长内存使用空间。

通过让索引和记录共享一个环形缓冲区, 可舍弃io.sort.record.percent参数, 这样, 不仅解决了用户设置参数的苦恼, 也使得Map Task能够最大限度地利用io.sort.mb空间, 进而减少磁盘溢写次数, 提高效率。

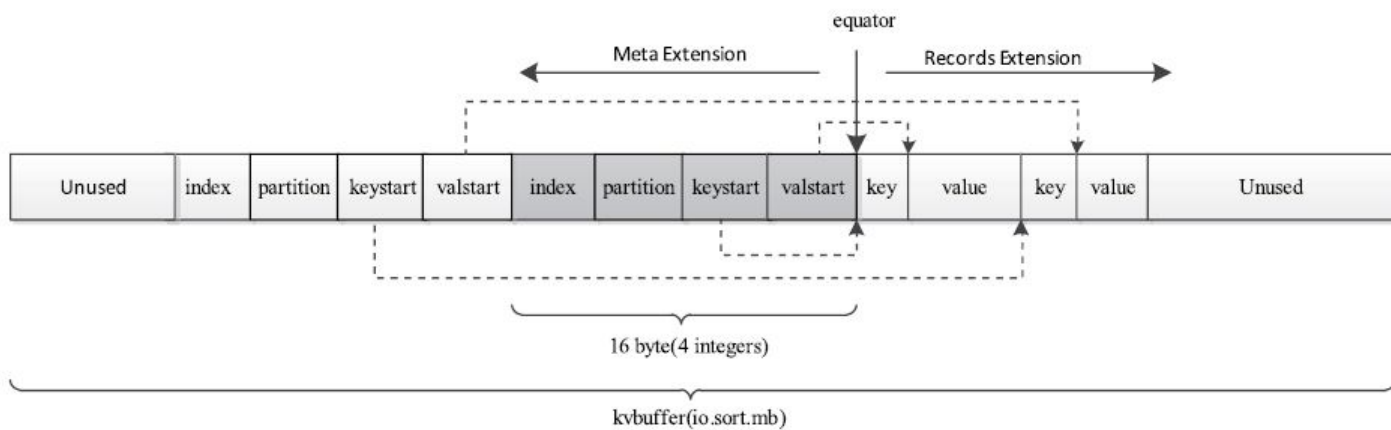


图 8-21 Hadoop 0.21 中 Map Task 的环形缓冲区结构

[1] Hadoop也曾采用过双缓冲区，具体可参考：<https://issues.apache.org/jira/browse/HADOOP-1965>。

[2] Todd Lipcon, Cloudera, “Optimiziong MapReduce Job Performance”, Hadoop Summit 2012.

[3] <https://issues.apache.org/jira/browse/MAPREDUCE-64>

8.3.3 Spill过程分析

Spill过程由SpillThread线程完成。在前一小节中已经提到，SpillThread线程实际上是缓冲区kvbuffer的消费者，其主要代码如下：

```
spillLock.lock();
while (true) {
    spillDone.signal();
    while (kvstart==kvend) {
        spillReady.await();
    }
    spillLock.unlock();
    sortAndSpill(); //排序，然后将缓冲区kvbuffer中的数据写到磁盘上
    spillLock.lock();
    //重置各个指针，以便为下一次溢写做准备
    if (bufend<bufindex&&bufindex<bufstart) {
        bufvoid=kvbuffer.length;
    }
    vstart=kvend;
    bufstart=bufend;
}
spillLock.unlock();
```

线程SpillThread调用函数sortAndSpill()将环形缓冲区kvbuffer中区间[bufstart, bufend)内的数据写到磁盘上。函数sortAndSpill()内部工作流程如下：

步骤1 利用快速排序算法对缓冲区kvbuffer中区间[bufstart, bufend)内的数据进行排序，排序方式是，先按照分区编号partition进行排序，然后按照key进行排序。这样，经过排序后，数据以分区为单位聚集在一起，且同一分区内所有数据按照key有序。

步骤2 按照分区编号由小到大依次将每个分区中的数据写入任务工作目录下的临时文件output/spillN.out（N表示当前溢写次数）中。如果用户设置了Combiner，则写入文件之前，对每个分区中的数据进行一次聚集操作。

步骤3 将分区数据的元信息写到内存索引数据结构SpillRecord中，其中每个分区的元信息包括在临时文件中的偏移量、压缩前数据大小和压缩后数据大小。如果当前内存中索引大小超过1 MB，则将内存索引写到文件output/spillN.out.index中。

8.3.4 Combine过程分析

当所有数据处理完后，Map Task会将所有临时文件合并成一个大文件，并保存到文件`output/file.out`中，同时生成相应的索引文件`output/file.out.index`。

在进行文件合并过程中，Map Task以分区为单位进行合并。对于某个分区，它将采用多轮递归合并的方式：每轮合并`io.sort.factor`（默认为100）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

让每个Map Task最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销^[1]。

^[1] <https://issues.apache.org/jira/browse/HADOOP-331>

8.4 Reduce Task内部实现

与Map Task一样，Reduce Task也分为四种，即Job-setup Task, Job-cleanup Task, Task-cleanup Task和Reduce Task。本节中重点介绍第四种——普通Reduce Task。

Reduce Task要从各个Map Task上读取一片数据，经排序后，以组为单位交给用户编写的reduce()函数处理，并将结果写到HDFS上。本节将深入剖析Reduce Task内部各个阶段的实现原理。

8.4.1 Reduce Task整体流程

Reduce Task的整体计算流程如图8-22所示，共分为5个阶段。

- ❑ Shuffle阶段：也称为Copy阶段。Reduce Task从各个Map Task上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。
- ❑ Merge阶段：在远程拷贝数据的同时，Reduce Task启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。
- ❑ Sort阶段：按照MapReduce语义，用户编写的reduce()函数输入数据是按key进行聚集的一组数据。为了将key相同的数据聚在一起，Hadoop采用了基于排序的策略。由于各个Map Task已经实现对自己的处理结果进行了局部排序，因此，Reduce Task只需对所有数据进行一次归并排序即可。
- ❑ Reduce阶段：在该阶段中，Reduce Task将每组数据依次交给用户编写的reduce()函数处理。
- ❑ Write阶段：reduce()函数将计算结果写到HDFS上。

在接下来几小节中，我们将详细介绍Shuffle、Merge、Sort和Reduce四个阶段。考虑到Write阶段比较简单，我们不再介绍。

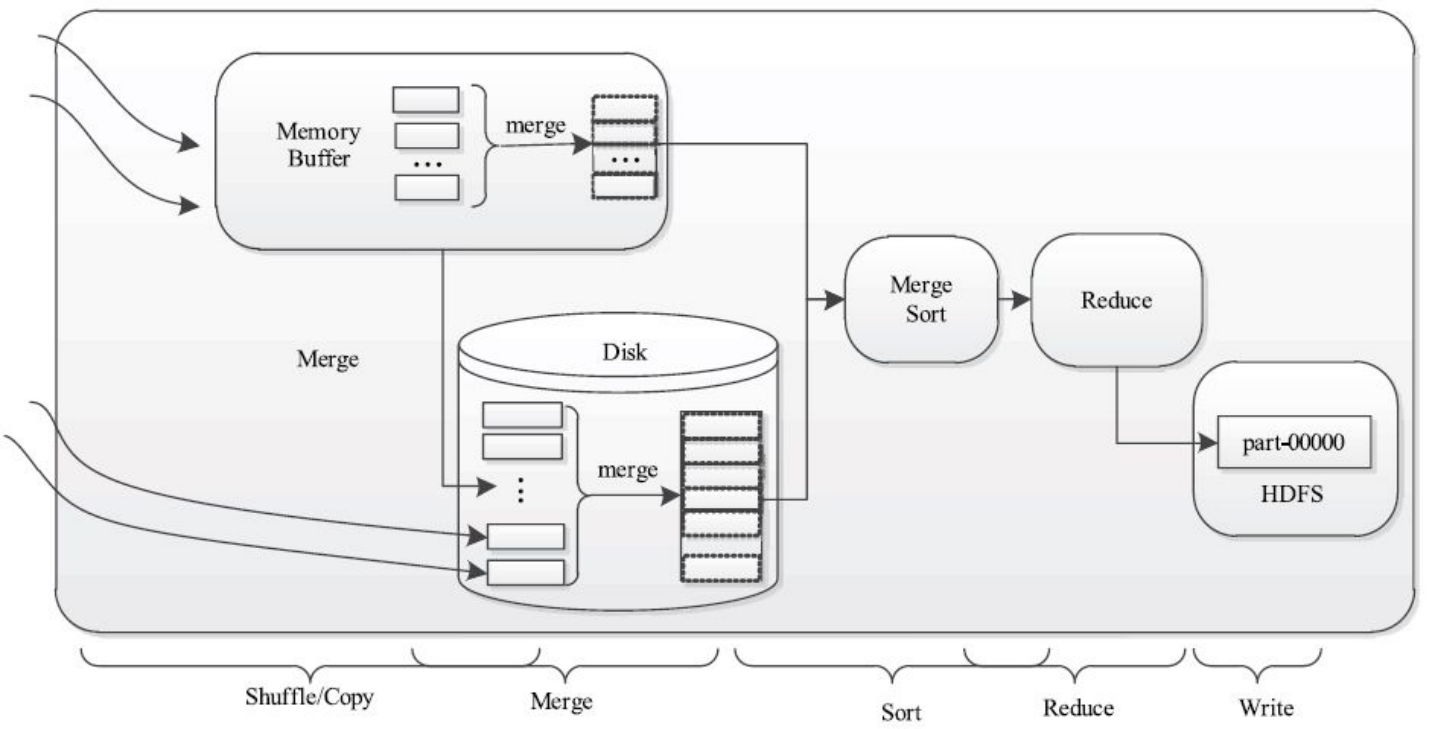


图 8-22 Map Task计算流程

8.4.2 Shuffle和Merge阶段分析

在Reduce Task中，Shuffle阶段和Merge阶段是并行进行的。当远程拷贝数据量达到一定阈值后，便会触发相应的合并线程对数据进行合并。这两个阶段均是由类ReduceCopier实现的，该类大约包含2 200行代码（整个ReduceTask类才2 900行左右）。如图8-23所示，总体上看，Shuffle& Merge阶段可进一步划分为三个子阶段。

(1) 准备运行完成的Map Task列表

GetMapEventsThread线程周期性通过RPC从TaskTracker获取已完成Map Task列表，并保存到映射表mapLocations（保存了TaskTracker Host与已完成任务列表的映射关系）中。为防止出现网络热点，Reduce Task通过对所有TaskTracker Host进行“混洗”操作以打乱数据拷贝顺序，并将调整后的Map Task输出数据位置保存到scheduledCopies列表中。

(2) 远程拷贝数据

Reduce Task同时启动多个MapOutputCopier线程，这些线程从scheduledCopies列表中获取Map Task输出位置，并通过HTTP Get远程拷贝数据。对于获取的数据分片，如果大小超过一定阈值，则存放到磁盘上，否则直接放到内存中。

(3) 合并内存文件和磁盘文件

为了防止内存或者磁盘上的文件数据过多，Reduce Task启动了LocalFSMerger和InMemFSMergeThread两个线程分别对内存和磁盘上的文件进行合并。

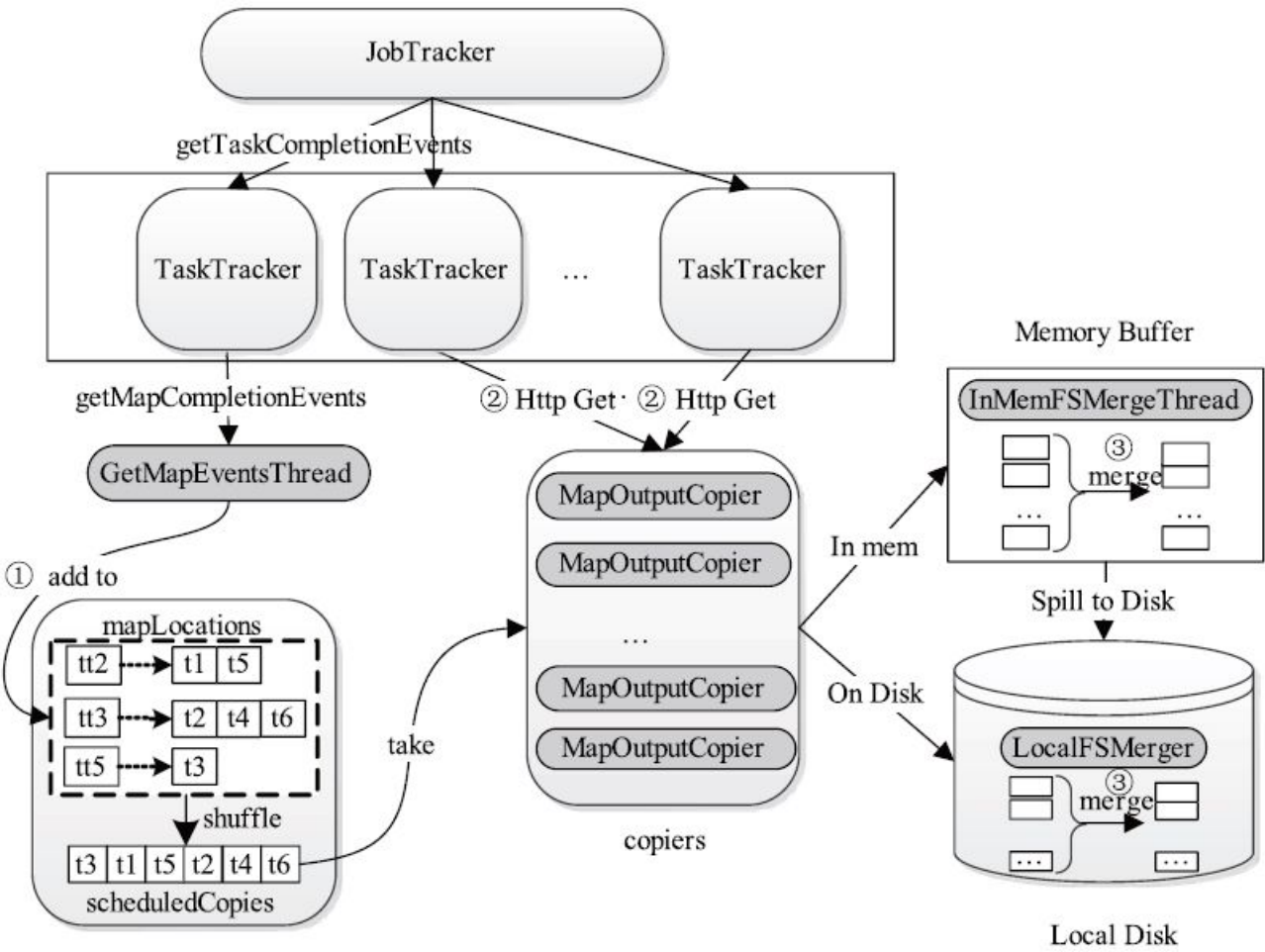


图 8-23 Shuffle阶段工作流程

接下来，我们将详细剖析每个阶段的内部实现细节。

（1）准备运行完成的Map Task列表

第7章讲到，TaskTracker启动了MapEventsFetcherThread线程。该线程会周期性（周期为心跳时间间隔）通过RPC从JobTracker上获取已经运行完成的Map Task列表，并保存到TaskCompletionEvent类型列表allMapEvents中。

而对于Reduce Task而言，它会启动GetMapEventsThread线程。该线程周期性通过RPC从TaskTracker上获取已运行完成的Map Task列表，并将成功运行完成的Map Task放到列表mapLocations中，具体如图8-24所示。

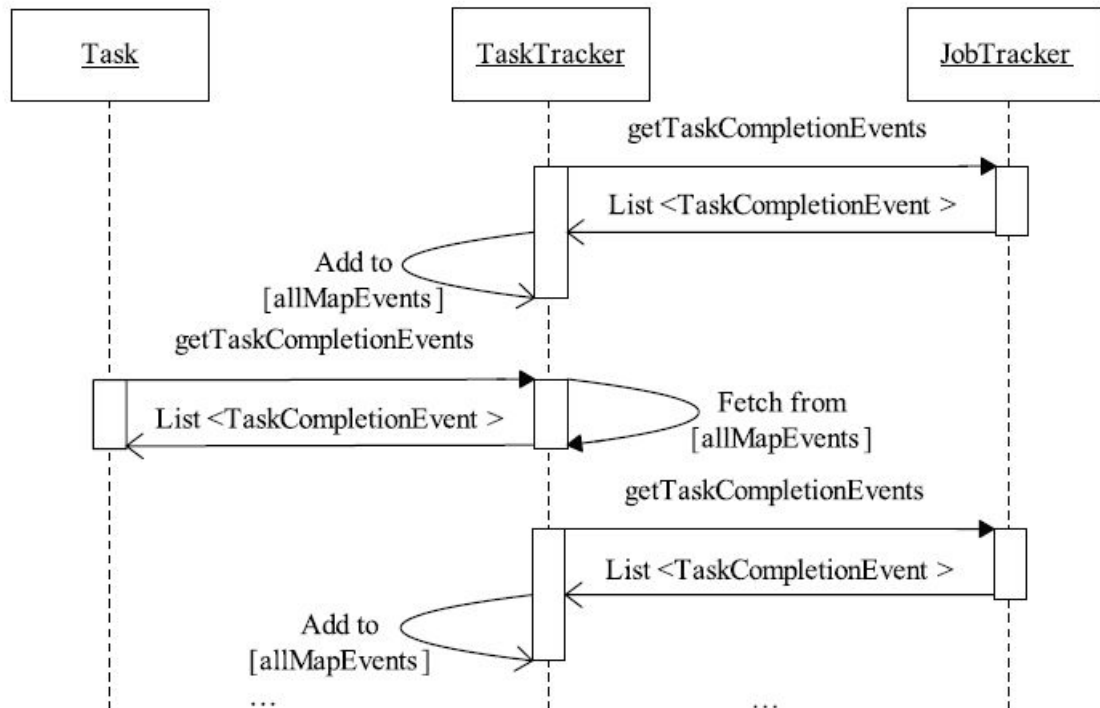


图 8-24 Reduce Task获取完成Map Task列表

为了避免出现数据访问热点（大量进程集中读取某个TaskTracker上的数据），Reduce Task不会直接将列表mapLocations中的Map Task输出数据位置交给MapOutputCopier线程，而是事先进行一次预处理：将所有TaskTracker Host进行混洗操作（随机打乱顺序），然后保存到scheduledCopies列表中，而MapOutputCopier线程将从该列表中获取待拷贝的Map Task输出数据位置。需要注意的是，对于一个TaskTracker而言，曾拷贝失败的Map Task将优先获得拷贝机会。

（2）远程拷贝数据

Reduce Task同时启动mapred.reduce.parallel.copies（默认是5）个数据拷贝线程MapOutputCopier。该线程从scheduledCopies列表中获取Map Task数据输出描述对象，并利用HTTP Get从对应的TaskTracker远程拷贝数据，如果数据分片大小超过一定阈值，则将数据临时写到工作目录下，否则直接保存到内存中。不管是保存到内存中还是磁盘上，MapOutputCopier均会保存一个MapOutput对象描述数据的元信息。如果数据被保存到内存中，则将该对象添加到列表mapOutputsFilesInMemory中，否则将该对象保存到列表mapOutputFilesOnDisk中。

在Reduce Task中，大部分内存用于缓存从Map Task端拷贝的数据分片，这些内存占到JVM Max Heap Size（由参数-Xmx指定）的mapred.job.shuffle.input.buffer.percent（默认是0.70）倍，并由类ShuffleRamManager管理。Reduce Task规定，如果一个数据分片大小未超过该内存的0.25倍，则可存放到内存中。如果MapOutputCopier线程要拷贝的数据分片可存放到内存中，则它先要向ShuffleRamManager申请相应的内存，待同意后才会正式拷贝数据，否则需要等待它释放内存。

由于远程拷贝数据可能需要跨网络读取多个节点上的数据，期间很容易由于网络或者磁盘等原因造成读取失败，因此提供良好的容错机制是非常有必要的。当出现拷贝错误时，Reduce Task提供了以下几个容错机制：

□如果拷贝数据出错次数超过`abortFailureLimit`，则杀死该Reduce Task（等待调度器重新调度执行），其中，`abortFailureLimit`计算方法如下：

$$\text{abortFailureLimit} = \max\{30, \text{numMaps}/10\}$$

□如果拷贝数据出错次数超过`maxFetchFailuresBeforeReporting`（可通过参数`mapreduce.reduce.shuffle.maxfetchfailures`设置，默认是10），则进行一些必要的检查^[1]，以决定是否杀死该Reduce Task。

□如果前两个条件均不满足，则采用对数回归模型推迟一段时间后重新拷贝对应MapTask的输出数据，其中延迟时间`delayTime`的计算方法如下：

$$\text{delayTime} = 10\,000 \times 1.3^{\text{noFailedFetches}}$$

其中`noFailedFetches`为拷贝错误次数。

（3）合并内存文件和磁盘文件

前面提到，Reduce Task从Map Task端拷贝的数据，可能保存到内存或者磁盘上。随着拷贝数据的增多，内存或者磁盘上的文件数目也必将增加，为了减少文件数目，在数据拷贝过程中，线程`LocalFSMerger`和`InMemFSMergeThread`将分别对内存和磁盘上的文件进行合并。

对于磁盘上文件，当文件数目超过 $(2 * \text{ioSortFactor} - 1)$ 后（`ioSortFactor`值由参数`io.sort.factor`指定，默认是10），线程`LocalFSMerger`会从列表`mapOutputFilesOnDisk`中取出最小的`ioSortFactor`个文件进行合并，并将合并后的文件再次写到磁盘上。

对于内存中的文件，当满足以下几个条件之一时，`InMemFSMergeThread`线程会将内存中所有数据合并后写到磁盘上：

□所有数据拷贝完毕后，关闭`ShuffleRamManager`。

□`ShuffleRamManager`中已使用内存超过可用内存的`mapred.job.shuffle.merge.percent`（默认是66%）倍且内存文件数目超过2个。

□内存中的文件数目超过`mapred.inmem.merge.threshold`（默认是1 000）。

□阻塞在`ShuffleRamManager`上的请求数目超过拷贝线程数目`mapred.reduce.parallel.copies`的75%。

^[1] 综合考虑Reduce Task拷贝失败的数据分片比例、拷贝成功的数据分片比例和最近来拷贝数据持续间隔等因素。

8.4.3 Sort和Reduce阶段分析

当所有数据拷贝完成后，数据可能存放在内存中或者磁盘上，此时还不能将数据直接交给用户编写的`reduce()`函数处理。根据MapReduce语义，Reduce Task需将key值相同的数据聚集到一起，并按组将数据交给`reduce()`函数处理。为此，Hadoop采用了基于排序的数据聚集策略。前面提到，各个Map Task已经事先对自己的输出分片进行了局部排序，因此，Reduce Task只需进行一次归并排序即可保证数据整体有序。为了提高效率，Hadoop将Sort阶段和Reduce阶段并行化。在Sort阶段，Reduce Task为内存和磁盘中的文件建立了小顶堆，保存了指向该小顶堆根节点的迭代器，且该迭代器保证了以下两个约束条件：

- 磁盘上文件数目小于`io.sort.factor`（默认是10）。

- 当Reduce阶段开始时，内存中数据量小于最大可用内存（JVM Max Heap Size）的`mapred.job.reduce.input.buffer.percent`（默认是0）。

在Reduce阶段，Reduce Task不断地移动迭代器，以将key相同的数据顺次交给`reduce()`函数处理，期间移动迭代器的过程实际上就是不断调整小顶堆的过程，这样，Sort和Reduce可并行进行。

8.5 Map/Reduce Task优化

对于任何一个作业，可从应用程序、参数和系统三个角度进行性能优化，其中，前两种需根据应用程序自身特点进行，而系统优化需从Hadoop平台设计缺陷出发进行系统级改进。本节重点介绍参数调优和系统优化两种方法。

8.5.1 参数调优

由于参数调优与应用程序的特点直接相关，因此本小节仅列出了Map Task和Reduce Task中直接影响任务性能的一些可调整参数（见表8-4和表8-5），具体调整为何值需由用户根据作业特点自行决定。

表 8-4 Map Task 可调整参数

参数名称	参数含义	默认值
io.sort.mb	Map Task 缓冲区所占内存大小	100 MB
io.sort.record.percent	缓冲 kvoffsets 和 kvindices 共占 io.sort.mb 的内存比例	0.05
io.sort.spill.percent	缓冲区 kvoffsets 或者 kvoffsets 内存使用率达到该比例后，会触发“溢写”操作，将内存中数据写成一个文件	0.80
mapred.compress.map.output	是否压缩 Map Task 中间输出结果	true
mapred.map.output.compression.codec	如果支持压缩 Map Task 中间结果，则采用什么压缩器	org.apache.hadoop.io.compress.zlib

表 8-5 Reduce Task 可调整参数

参数名称	参数含义	默认值
tasktracker.http.threads	HTTP Server 上的线程数。该 Server 运行在每个 TaskTracker 上，用于处理 Map Task 输出	40
mapred.reduce.parallel.copies	Reduce Task 同时启动的数据拷贝线程数目	5
mapred.job.shuffle.input.buffer.percent	ShuffleRamManager 管理的内存占 JVM Heap Max Size 的比例	0.70
mapred.job.shuffle.merge.percent	当内存使用率超过该值后，会触发一次合并，以将内存中的数据写到磁盘上	0.66
mapred.inmem.merge.threshold	当内存中的文件超过该阈值时，会触发一次合并，将内存中的数据写到磁盘上	1 000
io.sort.factor	文件合并时，一次合并的文件数目（合并后，将合并后的文件放到磁盘上继续合并，注意，每次合并时，选择最小的前 io.sort.factor 进行合并）	10 或 100 ^①
mapred.job.reduce.input.buffer.percent	Hadoop 假设用户的 reduce() 函数需要所有的 JVM 内存，因此执行 reduce() 函数前要释放所有内存。如果设置了该值，可将部分文件保存在内存中（不必写到磁盘上）	0

考虑到Hadoop中用户可配置参数非常多，为了简化参数配置，一些研究机构尝试自动调优参数，比较有名的是杜克大学的Starfish项目，有兴趣的读者可查看网站：<http://www.cs.duke.edu/starfish/>。

[1] Shuffle阶段文件合并时，该值默认为10；Sort阶段默认值是100。

8.5.2 系统优化

本小节主要讨论Hadoop的性能优化方向，但并不涉及对其架构进行大的调整或者改变其应用场景。

在Apache Hadoop中，Map/Reduce Task实现存在诸多不足之处，比如强制使用基于排序的聚集策略，Shuffle机制实现过于低效等。为此，下一代MapReduce提出了很多优化和改进方案，主要体现在以下几个方面^[1]。

1.避免排序^[2]

Hadoop采用了基于排序的数据聚集策略，而该策略是不可以定制的。也就是说，用户不可以使用其他数据聚集算法（如Hash聚集），也不可以跳过该阶段。而在实际应用中，很多应用可能不需要对数据进行排序，比如Hash join，或者基于排序的方法非常低效，比如SQL中的“limit-k”。为此，HDP版本^[3]提出将排序变成可选环节，这可带来以下几个方面的改进：

- ❑在Map Collect阶段，不再需要同时比较partition和key，只需比较partition，并可使用更快的计数排序（ $O(\lg N)$ ）代替快速排序（ $O(N \lg N)$ ）。
- ❑在Map Combine阶段，不再需要进行归并排序，只需按照字节合并数据块即可。
- ❑去掉排序后，Shuffle和Reduce可同时进行，即Reduce阶段可提前运行，这就消除了Reduce Task的屏障（所有数据拷贝完成后才能执行reduce()函数）。

2.Shuffle阶段内部优化

（1）Map端——用Netty代替Jetty

1.0.0版本中，TaskTracker采用了Jetty服务器处理来自各个Reduce Task的数据读取请求。由于Jetty采用了非常简单的网络模型，因此性能比较低。在Apache Hadoop 2.0.0版本中，Hadoop改用Netty，它是另一种开源的客户/服务器端编程框架。由于它内部采用了Java NIO技术，相比Jetty更加高效，且Netty社区更加活跃，其稳定性比Jetty好。

（2）Reduce端——批拷贝

1.0.0版本中，在Shuffle过程中，Reduce Task会为每个数据分片建立一个专门的HTTP连接（One-connection-per-map），即使多个分片同时出现在一个TaskTracker上，也是如此。为了提高数据拷贝效率，Apache Hadoop 2.0.0尝试采用批拷贝技术：不再为每个Map Task建立一个HTTP连接，而是为同一个TaskTracker上的多个Map Task建立一个HTTP连接，进而能够一次读取多个数据分片，具体如图8-25所示。

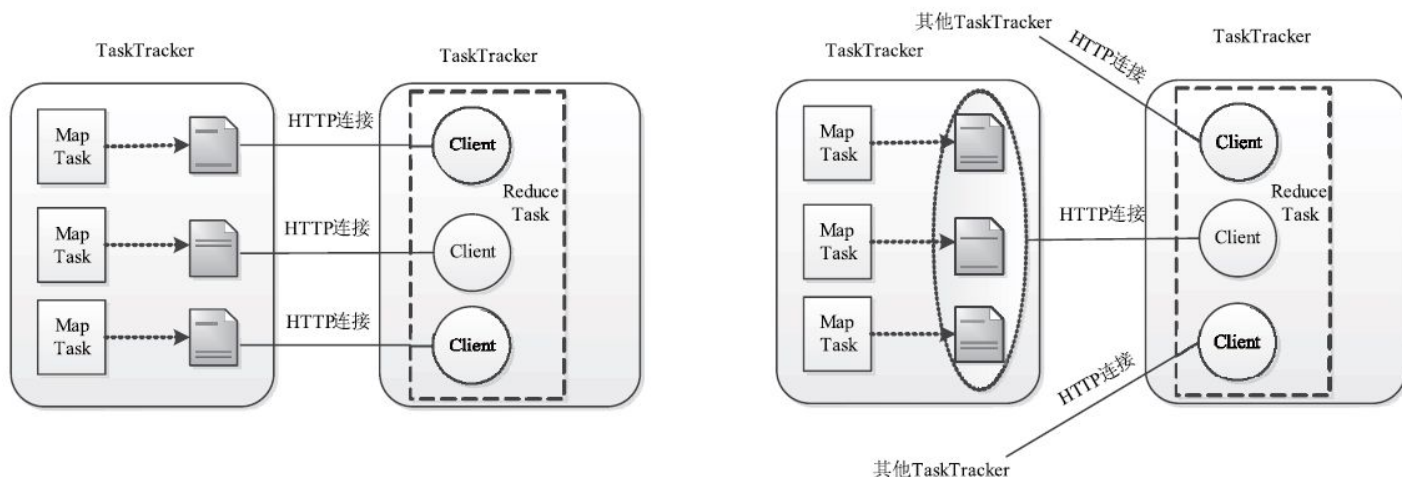


图 8-25 One-connection-per-map与批拷贝模型对比

3.将Shuffle阶段从Reduce Task中拆分出来

前面提到，对于一个作业而言，当一定比例（默认是5%）的Map Task运行完成后，Reduce Task才开始被调度，且仅当所有Map Task运行完成后，Reduce Task才可能运行完成。在所有Map Task运行完成之前，已经启动的Reduce Task将始终处于Shuffle阶段，此时它们不断从已经完成的Map Task上远程拷贝中间处理结果，由于随着时间推移，不断会有新的Map Task运行完成，因此Reduce Task会一直处于“等待—拷贝—等待—拷贝……”的状态。待所有Map Task运行完成后，Reduce Task才可能将结果全部拷贝过来，这时候才能够进一步调用用户编写的reduce()函数处理数据。从以上Reduce Task内部运行流程分析可知，Shuffle阶段会带来两个问题：slot Hoarding和资源利用率低下。

（1）Slot Hoarding现象

Slot Hoarding是一种资源囤积现象，具体表现是：对于任意一个MapReduce作业而言，在所有Map Task运行完成之前，已经启动的Reduce Task将一直占用着slot不释放。Slot Hoarding可能会导致一些作业产生饥饿现象。下面给出一个例子进行说明。

【实例】如图8-26所示，整个集群中有三个作业，分别是job1、job2和job3，其中，job1的Map Task数目非常多，而其他两个作业的Map Task相对较少。在t0时刻，job1和job2的Reduce Task开始被调度；在t3时刻，job2的所有Map Task运行完成，不久之后（t3'时刻），job2的第一批Reduce Task运行完成；在t4时刻，job2所有Reduce Task运行完成；在t4时刻，job3的Map Task开始运行并在t7时刻运行完成，但由于此时所有Reduce slot均被job1占用着，因此，除非job1的所有Map Task运行完成，否则job3的Reduce Task永远不可能得到调度。

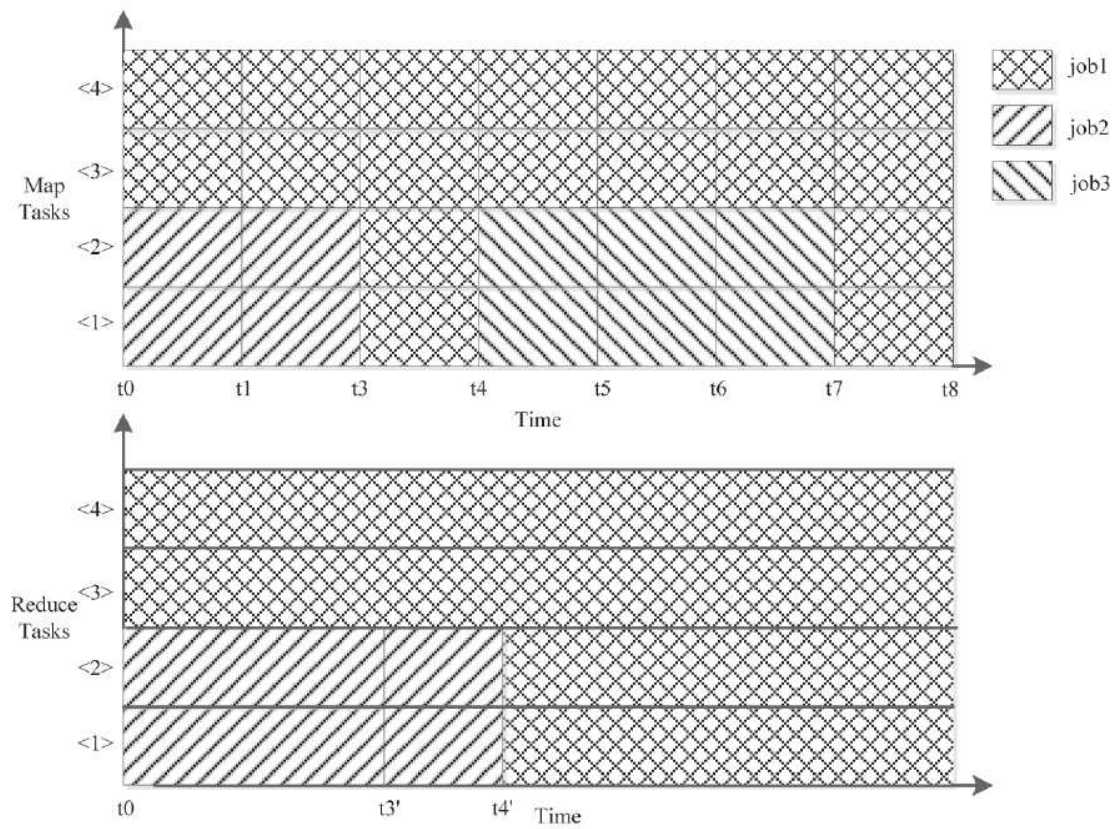


图 8-26 多个作业产生Slot Hoarding现象

(2) 资源利用率低下

从资源利用率角度看，为了保证较高的系统资源利用率，所有Task都应充分使用一个slot所隐含的资源，包括内存、CPU、I/O等资源。然而，对单个Reduce Task而言，在整个运行过程中，它的资源利用率很不均衡，总体上看，刚开始它主要使用I/O资源（Shuffle阶段），之后主要使用CPU资源（Reduce阶段）。如图8-27所示，t4时刻之前，所有已经启动的Reduce Task处于Shuffle阶段，此时主要使用网络I/O和磁盘I/O资源，而在t4时刻之后，所有Map Task运行完成，则第一批Reduce Task逐渐开始进入Reduce阶段，此时主要消耗CPU资源。由此可见，Reduce Task运行过程中使用的资源依次以I/O、CPU为主，并没有重叠使用这两种资源，这使得系统整体资源利用率低下。

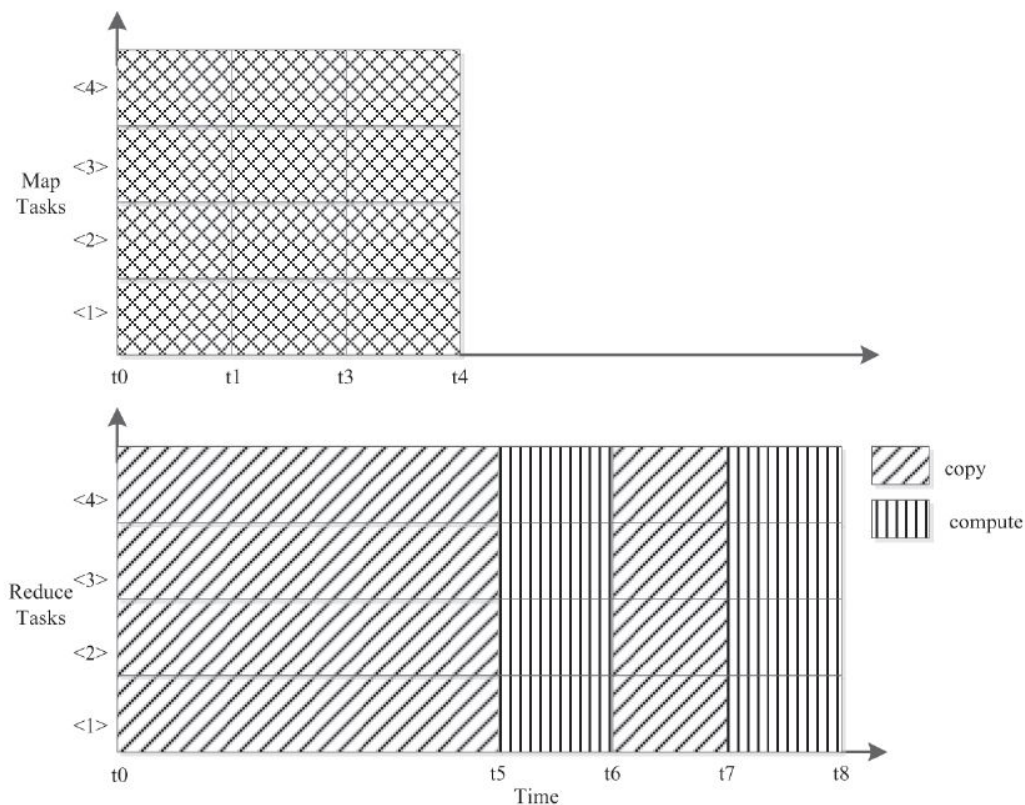


图 8-27 单个作业的Reduce Task资源利用率分析

经过以上分析可知，I/O密集型的数据拷贝（Shuffle阶段）和CPU密集型的数据计算（Reduce阶段）紧耦合在一起是导致“Slot Hoarding”现象和系统资源利用率低下的主要原因。为了解决该问题，一种可行的解决方案是将Shuffle阶段从Reduce Task中分离出来，当前主要有以下两种具体的实现方案。

□ **Copy-Compute Splitting**: 这是Berkeley的一篇论文^[4]提出的解决方案。该方案从逻辑上将Reduce Task拆分成“Copy Task”和“Compute Task”，其中，Copy Task用于数据拷贝，而Compute Task用于数据计算（调用用户编写的reduce()函数处理数据）。当一个Copy Task运行完成后，它会触发一个Compute Task进行数据计算，同时另外一个Copy Task将被启动拷贝另外的数据，从而实现I/O和CPU资源重叠使用。

□ **将Shuffle阶段变为独立的服务**: 将Shuffle阶段从Reduce Task处理逻辑中出来变成一个独立的服务，不再让其占用Reduce slot，这样也可达到I/O和CPU资源重叠使用的目的。“百度”曾采用了这一方案^[5]。

4.用C++改写Map/Reduce Task

利用C++实现Map/Reduce Task可借助C++语言独特的优势提高输出性能。当前比较典型的实现是NativeTask^[6]。NativeTask是一个C++实现的高性能MapReduce执行单元，它专注于数据处理本身。在MapReduce的环境下，它仅替换Task模块功能。也就是说，NativeTask并不关心资源管理、作业调度和容错等，这些功能仍旧由原有的Hadoop相应模块完成，而实际的数据处理则改由这个高性能处理引擎完成。

与Hadoop MapReduce相比，NativeTask获得了不错的性能提升，主要包括更好的排序实现、关键路径避免序列化、避免复杂抽象、更好地利用压缩等。

[1] 本节主要讨论Hadoop的性能优化方法，这些方法不会对其进行大的调整，比如改变其架构或者应用场景（离线改为在线）。

[2] <https://issues.apache.org/jira/browse/MAPREDUCE-4039>

[3] <https://github.com/hanborq/hadoop>

[4] M. Zaharia, D.Borthakur, J.S.Sarma, K.Elmeleggy, S.Shenker, and I.Stoica, “Job scheduling for multi-user mapreduce clusters,”EECS Department, University of California, Berkeley, Tech.Rep., Apr 2009.

[5] 连林江:“百度分布式计算技术发展”, PPT, 2012.07.08.

[6] <https://github.com/decster/nativetask>

8.6 小结

本章通过将任务进行阶段细分，详细介绍了Map Task和Reduce Task内部实现原理。

本章将Map Task分解成Read、Map、Collect、Spill和Combine五个阶段，并详细介绍了后三个阶段：map()函数处理完结果后，Map Task会将处理结果存放到一个内存缓冲区中（Collect阶段），待缓冲区使用率达到一定阈值后，再将数据溢写到磁盘上（Spill阶段），而当所有数据处理完后，Map Task会将磁盘上所有文件合并成一个大文件（Combine阶段）。这几个阶段形成的流水线如图8-28所示。

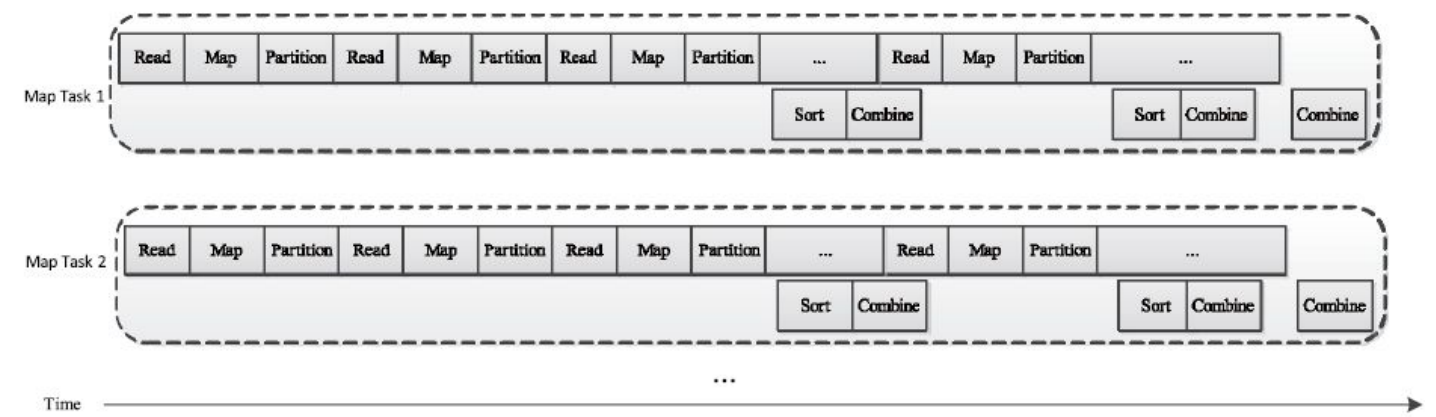


图 8-28 Map Task内部流水线

本章将Reduce Task分解成Shuffle、Merge、Sort、Reduce和Write五个阶段，且重点介绍了前三个阶段：Reduce Task首先进入Shuffle阶段，在该阶段中，它会启动若干个线程，从各个完成的Map Task上拷贝数据，并将数据放到磁盘上或者内存中，待文件数目超过一定阈值后进行一次合并（Merge阶段），当所有数据拷贝完成后，再对所有数据进行一次排序（Sort阶段），并将key相同的记录分组依次交给reduce()函数处理。这几个阶段形成的流水线如图8-29所示。

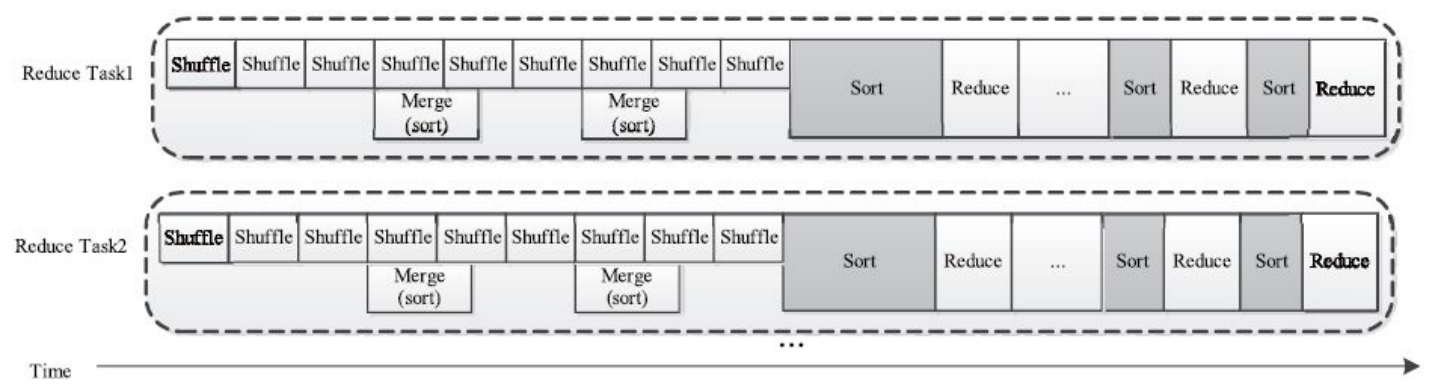


图 8-29 Reduce Task内部流水线

本章最后从参数调优和系统优化两个角度介绍了MapReduce作业优化方法。

本部分内容

Hadoop性能调优

Hadoop多用户作业调度器

Hadoop安全机制

下一代MapReduce框架

第9章 Hadoop性能调优

在前几章中，我们深入介绍了Hadoop MapReduce内部实现原理，包括JobTracker、TaskTracker、Task等组件的实现细节。基于对这些组件的深入理解，用户可以很容易通过调整一些关键参数使作业运行效率达到最优。本章将分别从Hadoop管理员和用户角度介绍如何对Hadoop进行性能调优以满足各自的需求。本章中的大部分内容在前几章中已经有所涉及，如果你对Hadoop优化方法已经非常熟悉，可以跳过本章。

9.1 概述

Hadoop性能调优是一项工程浩大的工作，它不仅涉及Hadoop本身的性能调优，还涉及更底层的硬件、操作系统和Java虚拟机等系统的调优，具体如图9-1所示。对这几个系统适当地进行调优均有可能给Hadoop带来性能提升。

对于非Hadoop自身方面的性能调优，比如硬件（如CPU类型、内存大小的选择等）、操作系统（比如文件系统选择、IO Scheduler选择、启用预读取机制、关闭swap等）、Java虚拟机（比如调整JVM参数）等，市面上有很多书籍和技术文档已经进行了详细介绍，本章只是进行简单介绍 [1]。本章将重点讲解如何通过调整Hadoop自带的一些参数使作业运行效率达到最优。总体来说，提高作业运行效率需要Hadoop管理员和作业拥有者共同的努力，其中，管理员负责为用户提供一个高效的作业运行环境，而用户则负责根据自己作业的特点让它尽可能快地运行完成。

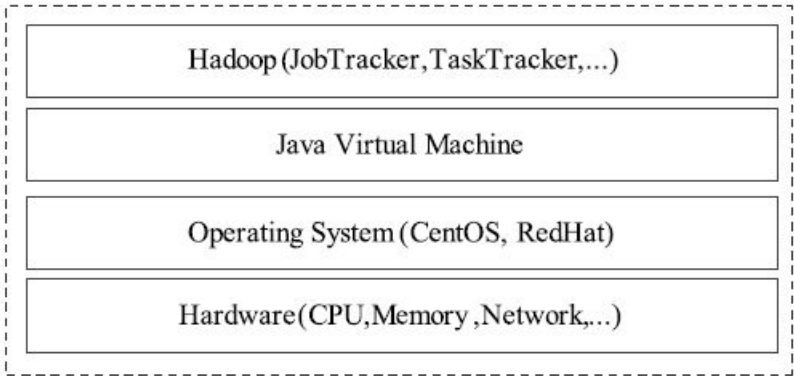


图 9-1 Hadoop层次结构图

在本书编写时，Apache Hadoop主要分为0.20.X（包括1.X）、0.21.X、0.22.X和0.23.X（包括2.0以上版本）四个系列，Cloudera Hadoop主要分为CDH 3和CDH 4两个系列，其中0.23.X（包括2.0以上版本）和CDH 4属于下一代MapReduce，本书不予讨论。这里重点介绍如何对Apache和Cloudera版本的第一代MapReduce进行性能调优。

[1] 参考AMD的技术文档“Hadoop Performance Tuning Guide”和Eric Sammer的书籍《Hadoop Operations》。

9.2 从管理员角度进行调优

管理员负责为用户作业提供一个高效的运行环境。管理员需要从全局出发，通过调整一些关键参数值提高系统的吞吐率和性能。总体上看，管理员需从硬件选择、操作系统参数调优、JVM参数调优和Hadoop参数调优等四个方面入手，为Hadoop用户提供一个高效的作业运行环境。

9.2.1 硬件选择

Hadoop自身架构的基本特点决定了其硬件配置的选型。Hadoop采用了master/slave架构，其中，master（JobTracker或者NameNode）维护了全局元数据信息，重要性远远大于slave（TaskTracker或者DataNode）。在较低Hadoop版本中，master均存在单点故障问题，因此，master的配置应远远好于各个slave（TaskTracker或者DataNode），具体可参考Eric Sammer的《Hadoop Operations》一书。

9.2.2 操作系统参数调优

由于Hadoop自身的一些特点，它只适合用于将Linux作为操作系统的生产环境。在实际应用场景中，管理员适当对Linux内核参数进行调优，可在一定程度上提高作业的运行效率，比较有用的调整选项如下。

（1）增大同时打开的文件描述符和网络连接上限

在Hadoop集群中，由于涉及的作业和任务数目非常多，对于某个节点，由于操作系统内核在文件描述符和网络连接数目等方面的限制，大量的文件读写操作和网络连接可能导致作业运行失败，因此，管理员在启动Hadoop集群时，应使用`ulimit`命令将允许同时打开的文件描述符数目上限增大至一个合适的值，同时调整内核参数`net.core.somaxconn`至一个足够大的值。

此外，Hadoop RPC采用了`epoll`作为高并发库，如果你使用的Linux内核版本在2.6.28以上，你需要适当调整`epoll`的文件描述符上限。

（2）关闭swap分区

在Linux中，如果一个进程的内存空间不足，那么，它会将内存中的部分数据暂时写到磁盘上，当需要时，再将磁盘上的数据动态置换到内存中，通常而言，这种行为会大大降低进程的执行效率。在MapReduce分布式计算环境中，用户完全可以通过控制每个作业处理的数据量和每个任务运行过程中用到的各种缓冲区大小，避免使用swap分区。具体方法是调整`/etc/sysctl.conf`文件中的`vm.swappiness`参数。

（3）设置合理的预读取缓冲区大小

磁盘I/O性能的发展远远滞后于CPU和内存，因而成为现代计算机系统的一个主要瓶颈。预读可以有效地减少磁盘的寻道次数和应用程序的I/O等待时间，是改进磁盘读I/O性能的重要优化手段之一。管理员可使用Linux命令`blockdev`设置预读取缓冲区的大小，以提高Hadoop中大文件顺序读的性能。当然，也可以只为Hadoop系统本身增加预读缓冲区大小，具体参考9.2.4节。

（4）文件系统选择与配置

Hadoop的I/O性能很大程度上依赖于Linux本地文件系统的读写性能。Linux中有多种文件系统可供选择，比如`ext3`和`ext4`，不同的文件系统性能有一定的差别。如果公司内部有自主研发的更高效的文件系统，也鼓励使用。

在Linux文件系统中，当未启用`noatime`属性时，每个文件读操作会触发一个额外的文件写操作以记录文件最近访问时间。该日志操作可通过将其添加到`mount`属性中避免。

（5）I/O调度器选择

主流的Linux发行版自带了很多可供选择的I/O调度器。在数据密集型应用中，不同的I/O调度器性能表现差别较大，管理员可根据自己的应用特点启用最合适的I/O调度器，具体可参考AMD的白皮书《Hadoop Performance Tuning Guide》。

除了以上几个常见的Linux内核调优方法外，还有一些其他的方法，管理员可根据需要进行适当调整。

9.2.3 JVM参数调优

由于Hadoop中的每个服务和任务均会运行在一个单独的JVM中，因此，JVM的一些重要参数也会影响Hadoop性能。管理员可通过调整JVM FLAGS和JVM垃圾回收机制提高Hadoop性能，具体可参考AMD的白皮书《Hadoop Performance Tuning Guide》。

9.2.4 Hadoop参数调优

1.合理规划资源

(1) 设置合理的槽位数目

在Hadoop中，计算资源是用槽位（slot）表示的。slot分为两种：Map slot和Reduce slot。每种slot代表了一定量的资源，且同种slot（比如Map slot）是同质的，也就是说，同种slot代表的资源量是相同的。管理员需根据实际需要为TaskTracker配置一定数目的Map slot和Reduce slot数目，从而限制每个TaskTracker上并发执行的Map Task和Reduce Task数目。

槽位数目是在各个TaskTracker上的mapred-site.xml中配置的，具体如表9-1所示。

表 9-1 设置槽位数目

Hadoop 版本号	配置参数	默认值
0.20.X（包括 1.X），CDH 3	mapred.tasktracker.map.tasks.maximum mapred.tasktracker.reduce.tasks.maximum	2 (两个参数值相同)
0.21.X, 0.22.X	mapreduce.tasktracker.map.tasks.maximum mapreduce.tasktracker.reduce.tasks.maximum	2 (两个参数值相同)

(2) 编写健康监测脚本

Hadoop允许管理员为每个TaskTracker配置一个节点健康状况监测脚本^[1]。TaskTracker中包含一个专门的线程周期性执行该脚本，并将脚本执行结果通过心跳机制汇报给JobTracker。一旦JobTracker发现某个TaskTracker的当前状况为“不健康”（比如内存或者CPU使用率过高），则会将其加入黑名单，从此不再为它分配新的任务（当前正在执行的任务仍会正常执行完毕），直到该脚本执行结果显示为“健康”。健康监测脚本的编写和配置的具体方法，可参考7.3.1节。

需要注意的是，该机制只有Hadoop 0.20.2以上版本中有。

2.调整心跳配置

(1) 调整心跳间隔

TaskTracker与JobTracker之间的心跳间隔大小应该适度。如果太小，JobTracker需要处理高并发的心跳信息，势必造成不小的压力；如果太大，则空闲的资源不能及时通知JobTracker（进而为之分配新的Task），造成资源空闲，进而降低系统吞吐率。对于中小规模（300个节点以下）的Hadoop集群，缩短TaskTracker与JobTracker之间的心跳间隔可明显提高系统吞吐率。

在Hadoop 1.0以及更低版本中，当节点集群规模小于300个节点时，心跳间隔将一直是3秒（不能修改）。这意味着，如果你的集群有10个节点，那么JobTracker平均每秒只需处理3.3（10/3=3.3）个心跳请求；而如果你的集群有100个节点，那么JobTracker平均每秒也只需处理33个心跳请求。对于一台普通的服务器，这样的负载过低，完全没有充分利用服务器资源。综上所述，对于中小规模的Hadoop集群，3秒的心跳间隔过大，管理员可根据需要适当减小心跳间隔^[2]，具体配置如表9-2所示。

表 9-2 设置心跳间隔

Hadoop 版本号	配置参数	默认值
0.20.X, 0.21.X, 0.22.X	不可配置	集群规模小于 300 时, 心跳间隔为 3 秒, 之后每增加 100 个节点, 则心跳间隔增加 1 秒
1.X, CDH 3	mapreduce.jobtracker.heartbeat.interval.min mapred.heartbeats.in.second mapreduce.jobtracker.heartbeats.scaling.factor	集群规模小于 300 时, 心跳间隔为 300 毫秒 (具体解释参考 6.3.2 小节)

(2) 启用带外心跳

通常而言, 心跳是由各个TaskTracker以固定时间间隔为周期发送给JobTracker的, 心跳中包含节点资源使用情况、各任务运行状态等信息。心跳机制是典型的pull-based模型。TaskTracker周期性通过心跳向JobTracker汇报信息, 同时获取新分配的任务。这种模型使得任务分配过程存在较大延时: 当TaskTracker出现空闲资源时, 它只能通过下一次心跳 (对于不同规模的集群, 心跳间隔不同, 比如1 000个节点的集群, 心跳间隔为10秒钟) 告诉JobTracker, 而不能立刻通知它。为了减少任务分配延迟, Hadoop引入了带外心跳 (out-of-band heartbeat) [3]。带外心跳不同于常规心跳, 它是任务运行结束或者任务运行失败时触发的, 能够在出现空闲资源时第一时间通知JobTracker, 以便它能够迅速为空闲资源分配新的任务。带外心跳的配置方法如表9-3所示。

表 9-3 配置带外心跳

Hadoop 版本号	配置参数	含义	默认值
0.20.2	未引入该机制	—	—
0.20.X (除 0.20.2), 0.21.X, 0.22.X, CDH 3	mapreduce.tasktracker.outofband.heartbeat	是否启用带外心跳	false

3.磁盘块配置

Map Task中间结果要写到本地磁盘上, 对于I/O密集型的任务来说, 这部分数据会对本地磁盘造成很大压力, 管理员可通过配置多块磁盘缓解写压力。当存在多块可用磁盘时, Hadoop将采用轮询的方式将不同Map Task的中间结果写到这些磁盘上, 从而平摊负载, 具体配置如表9-4所示。

表 9-4 配置多个磁盘块

Hadoop 版本号	配置参数	默认值
0.20.X (包括 1.X), CDH 3	mapred.local.dir	/tmp/hadoop-\${user.name}/mapred/local
0.21.X, 0.22.X	mapreduce.cluster.local.dir	/tmp/hadoop-\${user.name}/mapred/local

4.设置合理的RPC Handler和HTTP线程数目

(1) 配置RPC Handler数目

JobTracker需要并发处理来自各个TaskTracker的RPC请求, 管理员可根据集群规模和服务器并发处理能够调整RPC Handler数目, 以使JobTracker服务能力最佳, 配置方法如表9-5所示。

表 9-5 配置 RPC Handler 数目

Hadoop 版本号	配置参数	默认值
0.20.X (包括 1X), CDH 3	mapred.job.tracker.handler.count	10
0.21.X, 0.22.X	mapreduce.jobtracker.handler.count	10

(2) 配置HTTP线程数目

在Shuffle阶段，Reduce Task通过HTTP请求从各个TaskTracker上读取Map Task中间结果，而每个TaskTracker通过Jetty Server处理这些HTTP请求。管理员可适当调整Jetty Server的工作线程数以提高Jetty Server的并发处理能力，具体如表9-6所示。

表 9-6 配置 HTTP 线程数目

Hadoop 版本号	配置参数	默认值
0.20.X (包括 1.X), CDH 3	tasktracker.http.threads	40
0.21.X, 0.22.X	mapreduce.tasktracker.http.threads	40

5.慎用黑名单机制

当一个作业运行结束时，它会统计在各个TaskTracker上失败的任务数目。如果一个TaskTracker失败的任务数目超过一定值，则作业会将它加到自己的黑名单中。如果一个TaskTracker被一定数目的作业加入黑名单，则JobTracker会将该TaskTracker加入系统黑名单，此后JobTracker不再为其分配新的任务，直到一定时间段内没有出现失败的任务。

当Hadoop集群规模较小时，如果一定数量的节点被频繁加入系统黑名单中，则会大大降低集群吞吐率和计算能力，因此建议关闭该功能，具体配置方法可参考6.5.2小节。

6.启用批量任务调度

在Hadoop中，调度器是最核心的组件之一，它负责将系统中空闲的资源分配给各个任务。当前Hadoop提供了多种调度器，包括默认的FIFO调度器、Fair Scheduler、Capacity Scheduler等，调度器的调度效率直接决定了系统的吞吐率高低。通常而言，为了将空闲资源尽可能分配给任务，Hadoop调度器均支持批量任务调度^[4]，即一次将所有空闲任务分配下去，而不是一次只分配一个，具体配置如表9-7所示（FIFO调度器本身就是批量调度器）。

表 9-7 配置批量任务调度

调度器名称	Hadoop 版本	配置参数	参数含义	默认值
Capacity Scheduler	0.20.2, 0.21.X, 0.22.X	—	—	不支持批量调度，一次分配一个任务
	0.20.X（包括 1.X），CDH 3	mapred.capacity-scheduler.maximum-tasks-per-heartbeat	每次心跳最多分配的任务数目	32 767
Fair Scheduler	0.20.205 之前	—	—	不支持批量调度，一次分配一个任务
	0.20.205 之后, 0.21.X, 0.22.X	mapred.fairscheduler.assignmultiple mapred.fairscheduler.assignmultiple.maps mapred.fairscheduler.assignmultiple.reduce	是否启用批量调度功能，如果是，则一次最多分配的 Map Task 和 Reduce Task 数目	启用批量调度功能，且一次分配 Map Task 和 Reduce Task 的最高数目不受限

7.选择合适的压缩算法

Hadoop通常用于处理I/O密集型应用。对于这样的应用，Map Task会输出大量中间数据，这些数据的读写对用户是透明的，如果能够支持中间数据压缩存储，则会明显提升系统的I/O性能。当选择压缩算法时，需要考虑压缩比和压缩效率两个因素。有的压缩算法有很好的压缩比，但压缩/解压缩效率很低；反之，有一些算法的压缩/解压缩效率很高，但压缩比很低。因此，一个优秀的压缩算法需平衡压缩比和压缩效率两个因素。

当前有多种可选的压缩格式，比如gzip、zip、bzip2、LZO [5]、Snappy [6] 等，其中，LZO和Snappy在压缩比和压缩效率两方面的表现都比较优秀。其中，Snappy是Google开源的数据压缩库，它的编码/解码器已经内置到Hadoop 1.0以后的版本中 [7]；LZO则不同，它是基于GPL许可的，不能通过Apache来分发许可，因此，它的Hadoop编码/解码器必须单独下载 [8]。

下面以Snappy为例介绍如何让Hadoop压缩Map Task中间输出数据结果（在mapred-site.xml中配置）：

```
<property>
<name>mapred.compress.map.output</name>
<value>true</value>
</property>
<property>
<name>mapred.map.output.compression.codec</name>
<value>org.apache.hadoop.io.compress.SnappyCodec</value>
</property>
```

其中，“mapred.compress.map.output”表示是否要压缩Map Task中间输出结果，“mapred.map.output.compression.codec”表示采用的编码/解码器。

表9-8显示了Hadoop各版本是否内置了Snappy压缩算法。

表 9-8 配置 Snappy 压缩算法

Hadoop 版本号	是否内置 Snappy
0.20.X（不包括 1.X），0.21.X, 0.22.X	否
1.X, CDH 3	是

8.启用预读取机制

前面提到，预读取机制可以有效提高磁盘的I/O读性能。由于Hadoop是典型的顺序读系统，采用预读取机制可明显提高HDFS读

性能和MapReduce作业执行效率。管理员可为MapReduce的数据拷贝和IFile文件读取启用预读取功能 [9]，具体如表9-9所示。

表 9-9 配置预读取功能

Hadoop 版本号	配置参数	含 义	默认值
Apache 各版本和 CDH 3 u3 以下版本	暂未引入该机制	—	—
CDH 3 u3 以及更高版本	mapred.tasktracker.shuffle.fadvise	是否启用 Shuffle 预读取机制	true
	mapred.tasktracker.shuffle.readahead.bytes	Shuffle 预读取缓冲区大小	4 MB
	mapreduce.ifile.readahead	是否启用 IFile 预读取机制	true
	mapreduce.ifile.readahead.bytes	IFile 预读取缓冲区大小	4 MB

[1] <https://issues.apache.org/jira/browse/MAPREDUCE-211>

[2] <https://issues.apache.org/jira/browse/MAPREDUCE-1906>

[3] <https://issues.apache.org/jira/browse/MAPREDUCE-2355>

[4] <https://issues.apache.org/jira/browse/HADOOP-3136>

[5] <http://www.oberhumer.com/opensource/lzo/>

[6] <http://code.google.com/p/snappy/>

[7] <https://issues.apache.org/jira/browse/HADOOP-7206>

[8] <https://github.com/toddlipcon/hadoop-lzo>

[9] <https://issues.apache.org/jira/browse/HADOOP-7714>

9.3 从用户角度进行调优

Hadoop为用户作业提供了多种可配置的参数，以允许用户根据作业特点调整这些参数值使作业运行效率达到最优。

9.3.1 应用程序编写规范

尽管本章主要从参数配置方面介绍如何进行Hadoop调优，但从用户角度来看，除作业配置参数外，应用程序本身的编写方式对性能影响也是非常大的。在编写应用程序的过程中，谨记以下几条规则对提高作业性能是十分有帮助的。

（1）设置Combiner

对于一大批MapReduce应用程序，如果可以设置一个Combiner，那么对于提高作业性能是十分有帮助的。Combiner可减少Map Task中间输出结果，从而减少各个Reduce Task的远程拷贝数据量，最终表现为Map Task和Reduce Task执行时间缩短。

（2）选择合理的Writable类型

在MapReduce模型中，Map Task和Reduce Task的输入和输出数据类型均为Writable。Hadoop本身已经提供了很多Writable实现，包括IntWritable、FloatWritable。为应用程序处理的数据类型选择合适的Writable类型可大大提升性能。比如处理整型数据时，直接采用IntWritable比先以Text类型读入再转换成整型要高效。如果输出的整型大部分可用一个或者两个字节保存，那么可直接采用VIntWritable或者VLongWritable。它们采用了变长整型编码方式，可大大减少输出数据量。

9.3.2 作业级别参数调优

1.规划合理的任务数目

一个作业的任务数目对作业运行时间有重要的影响。如果一个作业的任务数目过多（这意味着每个任务处理数据很少，执行时间很短），则任务启动时间所占比例将会大大增加；反之，如果一个作业的任务数目过少（这意味着每个任务处理数据很多，执行时间很长），则可能会产生过多的溢写数据影响任务执行性能，且任务失败后重新计算代价过大。在Hadoop中，每个Map Task处理一个Input Split。Input Split的划分方式是由用户自定义的InputFormat决定的，默认情况下，由以下三个配置参数决定。

- mapred. min.split.size: Input Split的最小值（在mapred-site.xml中配置）。
- mapred. max.split.size: Input Split的最大值（在mapred-site.xml中配置）。
- dfs. block.size: HDFS中一个block大小（在hdfs-site.xml中配置）。

Map Task数目的具体算法可参考3.2.2小节。

对于Reduce Task而言，每个作业的Reduce Task数目通常由用户决定。用户可根据估算的Map Task输出数据量设置Reduce Task数目，以防止每个Reduce Task处理的数据量过大造成大量写磁盘操作。

2.增加输入文件副本数

如果一个作业并行执行的任务数量非常多，那么这些任务共同的输入文件可能成为瓶颈。为防止多个任务并行读取一个文件内容造成瓶颈，用户可根据需要增加输入文件的副本数目。用户可通过在客户端配置文件hdfs-site.xml中增加以下配置选项修改文件副本数，比如将客户端上传的所有数据副本数设置为5^[1]：

```
<property>
<name>dfs.replication</name>
<value>5</value>
</property>
```

3.启用推测执行机制

推测执行是Hadoop对“拖后腿”任务的一种优化机制。当一个作业的某些任务运行速度明显慢于同作业的其他任务时，Hadoop会在另一个节点上为“慢任务”启动一个备份任务，这样，两个任务同时处理一份数据，而Hadoop最终会将优先完成的那个任务的结果作为最终结果，并将另外一个任务杀掉。

用户可通过表9-10所示的参数设置是否为Map Task和Reduce Task启用推测执行机制。

表 9-10 启用推测执行机制

Hadoop 版本号	配置参数	默认值
0.20.X（包括 1.X），CDH 3	mapred.map.tasks.speculative.execution	true （两个参数值相同）
	mapred.reduce.tasks.speculative.execution	
0.21.X, 0.22.X	mapreduce.map.speculative	true （两个参数值相同）
	mapreduce.reduce.speculative	

不同Hadoop版本，采用的推测执行算法不同，具体可参考6.6节。

4.设置失败容忍度

Hadoop允许设置作业级别和任务级别的失败容忍度。作业级别的失败容忍是指Hadoop允许每个作业有一定比例的任务运行失败，这部分任务对应的输入数据将被忽略（这些数据不会有产出）；任务级别的失败容忍是指Hadoop允许任务运行失败后再次在另外节点

上尝试运行，如果一个任务经过若干次尝试运行后仍然运行失败，那么Hadoop才会最终认为该任务运行失败。

用户应根据应用程序的特点设置合理的失败容忍度，以尽快让作业运行完成和避免没必要的资源浪费，具体设置如表9-11所示。

表 9-11 设置失败容忍度

Hadoop 版本号	配置参数	参数含义	默认值
0.20.X（包括 1.X），CDH 3	mapred.max.map.failures.percent	作业最多允许失败的 Map Task 和 Reduce Task 比例	0（如果是 5，表示为 5%）
	mapred.max.reduce.failures.percent		
	mapred.map.max.attempts	一个 Map Task 或者 Reduce Task 最多尝试次数	4（两个参数值相同）
	mapred.reduce.max.attempts		
0.21.X，0.22.X	mapreduce.map.failures.maxpercent	作业最多允许失败的 Map Task 和 Reduce Task 比例	0（如果是 5，表示为 5%）
	mapreduce.reduce.failures.maxpercent		
	mapreduce.map.maxattempts	一个 Map Task 或者 Reduce Task 最多尝试次数	4（两个参数值相同）
	mapreduce.reduce.maxattempts		

5.适当打开JVM重用功能

为了实现任务隔离，Hadoop将每个任务放到一个单独的JVM中执行，而对于执行时间较短的任务，JVM启动和关闭将占用很大比例的时间，为此，用户可启用JVM重用功能，这样，一个JVM可连续启动多个同类型任务，具体可参考7.6.1节。设置JVM重用的方法如表9-12所示。

表 9-12 设置 JVM 重用

Hadoop 版本号	配置参数	默认值
0.20.X（包括 1.X），CDH 3	mapred.job.reuse.jvm.num.tasks	1 (1 表示每个 JVM 只能启动一个 Task；若为 -1，则表示每个 JVM 最多可运行的 Task 数目不受限制)
0.21.X，0.22.X	mapreduce.job.jvm.num.tasks	1

6.设置任务超时时间

在一些特殊情况下，一个任务可能因为某种原因（比如程序Bug，Hadoop本身的Bug等）阻塞了，这会拖慢整个作业的执行进度，甚至可能导致作业无法运行结束。针对这种情况，Hadoop增加了任务超时机制。如果一个任务在一定时间间隔内没有汇报进度，则TaskTracker会主动将其杀死，从而在另外一个节点上重新启动执行。

用户可根据实际需要配置任务超时时间，配置方法如表9-13所示。

表 9-13 设置任务超时时间

Hadoop 版本号	配置参数	默认值
0.20.X（包括 1.X），CDH 3	mapred.task.timeout	600 000（单位是毫秒，10 分钟）
0.21.X，0.22.X	mapreduce.task.timeout	600 000（单位是毫秒，10 分钟）

7.合理使用DistributedCache

当用户的应用程序需要一个外部文件（比如字典、配置文件等）时，通常需要使用DistributedCache将文件分发到各个节点上。一般情况下，得到外部文件有两种方法：一种是外部文件与应用程序jar包一起放到客户端，当提交作业时由客户端上传到HDFS的一个目录下，然后通过DistributedCache分发到各个节点上；另外一种方法是事先将外部文件直接放到HDFS上。从效率上讲，第二种方法比第一种更高效。第二种方式不仅节省了客户端上传文件的时间，还隐含着告诉DistributedCache：“请将文件下载到各节点的public级别

（而不是private级别）共享目录中”，这样，后续所有的作业可重用已经下载好的文件，不必重复下载，即“一次下载，终生受益”，具体参考5.4节。

8.合理控制Reduce Task的启动时机

在MapReduce计算模型中，由于Reduce Task依赖于Map Task的执行结果，因此，从运算逻辑上讲，Reduce Task应晚于Map Task启动。在Hadoop中，合理控制Reduce Task启动时机不仅可以加快作业运行速度，而且可提高系统资源利用率。如果Reduce Task启动过早，则可能由于Reduce Task长时间占用Reduce slot资源造成“slot Hoarding”现象（具体可参考8.5.2节），从而降低资源利用率；反之，如果Reduce Task启动过晚，则会导致Reduce Task获取资源延迟，增加了作业运行时间。Hadoop配置Reduce Task启动时机的参数如表9-14所示。

表 9-14 设置 Reduce Task 启动时机

Hadoop 版本号	配置参数	默认值
0.20.X（包括 1.X），CDH 3	mapred.reduce.slowstart.completed.maps	0.05 (Map Task 完成数目达到 5% 时，开始启动 Reduce Task)
0.21.X，0.22.X	mapreduce.job.reduce.slowstart.completed.maps	同上

9.跳过坏记录

Hadoop是用于处理海量数据的，对于大部分数据密集型应用而言，丢弃一条或者几条数据对最终结果的影响并不大，正因为如此，Hadoop为用户提供了跳过坏记录的功能。当一条或者几条坏数据记录导致任务运行失败时，Hadoop可自动识别并跳过这些坏记录，具体配置方法可参考6.5.4节。

10.提高作业优先级

所有Hadoop作业调度器进行任务调度时均会考虑作业优先级这一因素。一个作业的优先级越高，它能够获取的资源（指slot数目）也越多。需要注意的是，通常而言，在生产环境中，管理员已经按照作业重要程度对作业进行了分级，不同重要程度的作业允许配置的优先级不同，用户不可以擅自进行调整。Hadoop提供了5种作业优先级，分别是VERY_HIGH、HIGH、NORMAL、LOW和VERY_LOW。用户可在允许的范围内调整作业优先级以获取更多资源，可配置的参数如表9-15所示。

表 9-15 设置作业优先级

Hadoop 版本号	配置参数	默认值
0.20.X（包括 1.X），CDH 3	mapred.job.priority	NORMAL
0.21.X，0.22.X	mapreduce.job.priority	NORMAL

[1] 为了防止该参数对所有文件生效，可创建一个专门的配置文件仅供有需求的数据使用。另外，该参数只对参数修改之后上传的文件有效，而已经上传的文件副本数不会改变。

9.3.3 任务级别参数调优

1.Map Task调优

在8.3节中已经提到，Map Task的输出结果将被暂时存放到一个环形缓冲区中，这个缓冲区的大小由参数“`io.sort.mb`”指定（单位是MB，默认是100 MB）。该缓冲区主要由两部分组成：索引和实际数据。默认情况下，索引占整个buffer的比例为`io.sort.record.percent`（默认为0.05，即5%），剩下的空间全部存放数据，当且仅当满足以下任意一个条件时，才会触发一次flush，生成一个临时文件：

- 索引空间使用率达到比例为`io.sort.spill.percent`（默认是0.8，即80%）。

- 数据空间使用率达到比例为`io.sort.spill.percent`（默认是0.8，即80%）。

合理地调整`io.sort.record.percent`值，可减少中间文件数目，提高任务执行效率。举例说明，如果你的key/value非常小，则可以适当调大`io.sort.record.percent`值，以防止索引空间优先达到使用上限触发flush。考虑到每条数据记录（一个key/value）需占用索引大小为16 B，因此，建议 $\text{io.sort.record.percent} = 16 / (16 + R)$ ，其中R为平均每条记录的长度。

综上所述，用户可根据自己作业的特点对以下参数进行调优：

- `io.sort.mb`;

- `io.sort.record.percent`;

- `io.sort.spill.percent`。

2.Reduce Task调优

在8.3节中已经提到，Reduce Task会启动多个拷贝线程从每个Map Task上读取相应的中间结果，具体的拷贝线程数目由参数“`mapred.reduce.parallel.copies`”（默认为5）指定。对于每个待拷贝的文件，如果文件大小小于一定阈值A，则将其放到内存中，否则以文件的形式存放到磁盘上。如果内存中文件满足一定条件D，则会将这些数据写入磁盘，而当磁盘上文件数目达到`io.sort.factor`（默认是10）时，进行一次合并。阈值A为：

$$\text{heapsize} * \{\text{mapred.job.shuffle.input.buffer.percent}\} * 0.25$$

其中，`heapsize`是通过参数“`mapred.child.java.opts`”指定的，默认是200 MB；`mapred.job.shuffle.input.buffer.percent`默认大小为0.7。

条件D为以下两个条件中任意一个：

- 内存使用率（总的可用内存为 $\text{heapsize} * \{\text{mapred.job.shuffle.input.buffer.percent}\}$ ）达到`mapred.job.shuffle.merge.percent`（默认是0.66）。

- 内存中文件数目超过`mapred.inmem.merge.threshold`（默认是1 000）。

综上所述，用户可根据自己作业的特点对以下参数进行调优：

- `mapred.reduce.parallel.copies`;

- `io.sort.factor`;

- `mapred.child.java.opts`;

☐ mapred.job.shuffle.input.buffer.percent;

☐ mapred.inmem.merge.threshold。

9.4 小结

Hadoop性能调优是一项工程浩大的工作。它不仅涉及Hadoop本身的性能调优，还涉及更底层的硬件、操作系统和Java虚拟机等系统的调优。本章从硬件、操作系统、Java虚拟机和Hadoop参数调优等四个方面介绍了Hadoop性能调优的方法。

第10章 Hadoop多用户作业调度器

Hadoop最初是为批处理作业而设计的，当时仅采用了一个简单的FIFO调度机制分配任务。但随着Hadoop的普及，单个Hadoop集群中的用户量和应用程序种类不断增加，适用于批处理场景的FIFO调度机制不能很好地利用集群资源，也不能够满足不同应用程序的服务质量要求，因此，设计适用于多用户的作业调度器势在必行。

从目前看来，多用户作业调度器的设计思路主要有两种：第一种是在一个物理集群上虚拟多个Hadoop集群，这些集群各自拥有全套独立的Hadoop服务，比如JobTracker、TaskTracker等，典型代表是HOD（Hadoop On Demand）调度器；另一种是扩展Hadoop调度器，使之支持多个队列多用户，典型代表是Yahoo! 的Capacity Scheduler和Facebook的Fair Scheduler。本章将重点介绍这两种多用户作业调度器的应用场景和设计原理。

10.1 多用户调度器产生背景

Hadoop最初的设计目的是支持大数据批处理作业，如日志挖掘、Web索引等作业，为此，Hadoop仅提供了一个非常简单的调度机制：FIFO，即先来先服务。在该调度机制下，所有作业被统一提交到一个队列中，Hadoop按照提交顺序依次运行这些作业。

但随着Hadoop的普及，单个Hadoop集群的用户量越来越大，不同用户提交的应用程序往往具有不同的服务质量要求（Quality OfService, QoS），典型的应用有以下几种。

□批处理作业：这种作业往往耗时较长，对完成时间一般没有严格要求，如数据挖掘、机器学习等方面的应用程序。

□交互式作业：这种作业期望能及时返回结果，如SQL查询（Hive）等。

□生产性作业：这种作业要求有一定量的资源保证，如统计值计算、垃圾数据分析等。

此外，这些应用程序对硬件资源的需求量也是不同的，如过滤、统计类作业一般为CPU密集型作业，而数据挖掘、机器学习作业一般为I/O密集型作业。因此，传统的FIFO调度策略不仅不能满足多样化需求，也不能充分利用硬件资源。

为了克服单队列FIFO调度器的不足，多种类型的多用户多队列调度器诞生了。这种调度器允许管理员按照应用需求对用户或者应用程序分组，并为不同的分组分配不同的资源量，同时通过添加各种约束防止单个用户或者应用程序独占资源，进而能够满足各种QoS需求。当前主要有两种多用户作业调度器的设计思路：第一种是在一个物理集群上虚拟多个Hadoop集群，这些集群各自拥有全套独立的Hadoop服务，比如JobTracker、TaskTracker等，典型的代表是HOD调度器；另一种是扩展Hadoop调度器，使之支持多个队列多用户，这样，不同的队列拥有不同的资源量，可以运行不同的应用程序，典型的代表是Yahoo! 的Capacity Scheduler和Facebook的Fair Scheduler。接下来将分别介绍这两种多用户作业调度器。

10.2 HOD

HOD（Hadoop On Demand）调度器^[1]是一个在共享物理集群上管理若干个Hadoop集群的工具。用户可通过HOD调度器在一个共享物理集群上快速搭建若干个独立的虚拟Hadoop集群，以满足不同的用途，比如不同集群运行不同类型的应用程序，运行不同的Hadoop版本进行测试等。HOD调度器可使管理员和用户轻松地快速搭建和使用Hadoop。

HOD调度器首先使用Torque资源管理器^[2]为一个虚拟Hadoop集群分配节点，然后在分配的节点上启动MapReduce和HDFS中的各个守护进程，并自动为Hadoop守护进程和客户端生成合适的配置文件（包括mapred-site.xml, core-site.xml和hdfs-site.xml等）。接下来将分别介绍Torque资源管理器和HOD调度器的基本工作原理。

10.2.1 Torque资源管理器

HOD调度器的工作过程实现中依赖于一个资源管理器来为它分配、回收节点和管理各节点上的作业运行的情况，如监控作业的运行、维护作业的运行状态等。而HOD只需在资源管理器所分配的节点上运行Hadoop守护进程和MapReduce作业即可。当前HOD采用的资源管理器是开源的Torque资源管理器。

一个Torque集群由一个头节点和若干个计算节点组成。头节点上运行一个名为pbs_server的守护进程，主要用于管理计算节点和监控各个作业的运行状态。每个计算节点上运行一个名为pbs_mom的守护进程，用于执行主节点分配的作业。此外，用户可将任何节点作为客户端，用于提交和管理作业。

头节点内部还运行了一个调度器守护进程。该守护进程会与pbs_server进行通信，以决定对资源使用和作业分配的本地策略。默认情况下，调度守护进程采用了FIFO调度机制，它将所有作业存放到一个队列中，并按照到达时间依次对它们进行调度。需要注意的是，Torque中的调度机制是可插拔的，Torque还提供许多其他可选的作业调度器。

如图10-1所示，用户可通过qsub命令向物理集群中提交作业，而Torque内部执行流程如下：

- 步骤1 当pbs_server收到新作业后，会进一步通知调度器。
- 步骤2 调度器采用一定的策略为该作业分配节点，并将节点列表与节点对应的作业执行命令返回给pbs_server。
- 步骤3 pbs_server将作业发送给第一个节点。
- 步骤4 第一个节点启动作业，作业开始运行（该作业会通知其他节点执行相应命令）。
- 步骤5 作业运行完成或者资源申请到期后，Torque会回收资源。

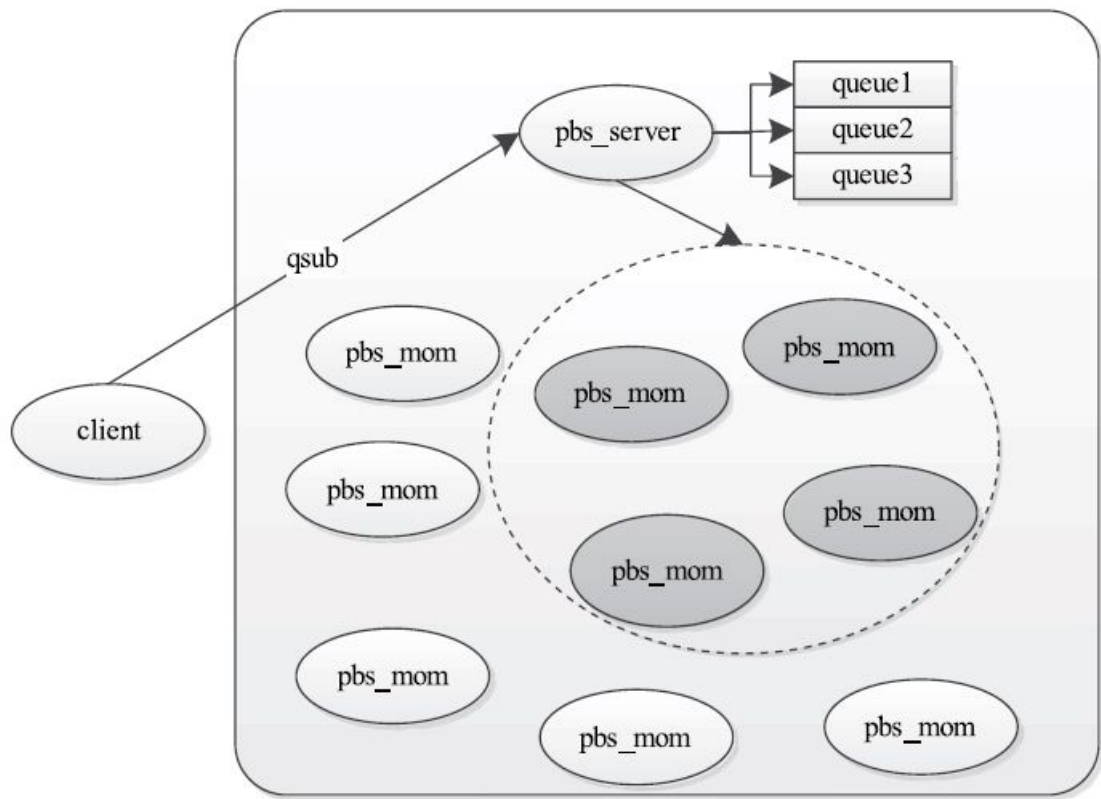


图 10-1 Torque内部工作原理

[1] http://hadoop.apache.org/docs/stable/hod_scheduler.html

[2] <http://www.adaptivecomputing.com/products/open-source/torque/>

10.2.2 HOD作业调度

理解了Torque工作原理后，HOD调度器工作原理便一目了然：首先利用Torque向物理集群申请一个虚拟机群，然后将Hadoop守护进程包装成一个Torque作业，并在申请的节点上启动，最后用户可直接向启动的Hadoop集群中提交作业。通过HOD调度器申请集群和运行作业的主要流程如下：

步骤1 用户向HOD调度器申请一个包含一定数目节点的集群，并要求该集群中运行一个Hadoop实例。

步骤2 HOD客户端利用资源管理器接口qsub提交一个被称为RingMaster的进程作为Torque作业，同时申请一定数目的节点。这个作业被提交到pbs_server上。

步骤3 在各个计算节点上，守护进程pbs_mom接受并处理pbs_server分配的作业。RingMaster进程在其中一个计算节点上开始运行。

步骤4 RingMaster通过Torque的另外一个接口pbsdsh在所有分配到的计算节点上运行第二个HOD组件HodRing，即运行于各个计算节点上的分布式任务。

步骤5 HodRing初始化之后会与RingMaster通信以获取Hadoop指令，并根据指令启动Hadoop服务进程。一旦服务进程开始启动，它们会向RingMaster登记，提供关于守护进程的信息。

注意 Hadoop实例所需的配置文件全部由HOD自己生成。HOD客户端保持和RingMaster的通信，可以获取MapReduce和HDFS守护进程所在的位置。

步骤6 Hadoop实例启动之后，用户可以向集群中提交MapReduce作业。

步骤7 如果一段时间内Hadoop集群上没有作业运行，Torque会回收该虚拟Hadoop集群的资源。

管理员将一个物理集群划分成若干个Hadoop集群后，用户可将不同类型的应用程序提交到不同Hadoop集群上，这样可避免不同用户或者不同应用程序之间争夺资源，从而达到多用户共享集群的目的。

从集群管理和资源利用率两方面看，这种基于完全隔离的集群划分方法存在诸多问题。

□从集群管理角度看，多个Hadoop集群会给运维人员造成管理上的诸多不便。

□多个Hadoop集群会导致集群整体利用率低下，这主要是负载不均衡造成的，比如某个集群非常忙碌时另外一些集群可能空闲，也就是说，多个Hadoop集群无法实现资源共享。

□考虑到虚拟集群回收后数据可能丢失，用户通常将虚拟集群中的数据写到外部的HDFS上。如图10-2所示，用户通常仅在虚拟集群上安装MapReduce，至于HDFS，则使用一个外部全局共享的HDFS。很明显，这种部署方法会导致丧失部分数据的本地特性。为了解决该问题，一种更好的方法是在整个集群中只保留一个Hadoop实例，而通过Hadoop调度器将整个集群中的资源划分给若干个队列，并让这些队列共享所有节点上的资源，当前Yahoo!的Capacity Scheduler和Facebook的Fair Scheduler正是采用了这个设计思路。

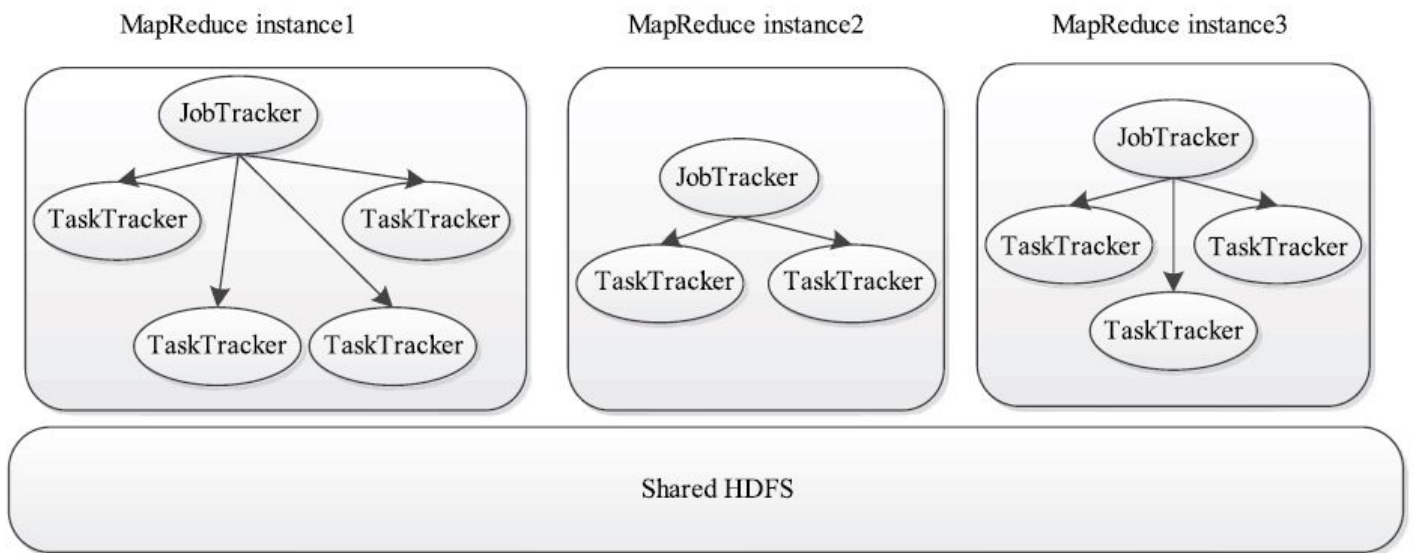


图 10-2 基于外部HDFS的多个虚拟MapReduce集群

10.3 Hadoop队列管理机制

在学习Capacity Scheduler和Fair Scheduler之前，我们先要了解Hadoop的用户和作业管理机制，这是任何Hadoop可插拔调度器的基础。

Hadoop以队列为单位管理作业、用户和资源，整个Hadoop集群被划分成若干个队列，每个队列被分配一定的资源，且用户只能向对应的一个或者几个队列中提交作业。Hadoop队列管理机制由用户权限管理和系统资源管理两部分组成，下面依次进行介绍。

1. 用户权限管理

Hadoop的用户管理模块构建在操作系统用户管理之上，增加了“队列”这一用户组织单元，并通过队列建立了操作系统用户和用户组之间的映射关系。管理员可配置每个队列对应的操作系统用户和用户组（需要注意的是，Hadoop允许一个操作系统用户或者用户组对应一个或者多个队列），也可以配置每个队列的管理员。他可以杀死该队列中任何作业，改变任何作业的优先级等。

Hadoop集群中所有队列需在配置文件mapred-site.xml中设置，这意味着该配置信息不可以动态加载。

【实例】如果一个集群中有四个队列，分别是queueA、queueB、queueC和default，那么可以在mapred-site.xml中配置如下：

```
<property>
<name>mapred.queue.names</name>
<value>queueA, queueB, queueC, default</value>
<description>Hadoop中所有队列名称</description>
</property>
<property>
<name>mapred.acls.enabled</name>
<value>true</value>
<description>是否启用权限管理功能</description>
</property>
```

队列权限相关的配置选项在配置文件mapred-queue-acls.xml中设置，这些信息可以动态加载。

【实例】如果规定用户linux_userA和用户组linux_groupA可以向队列queueA中提交作业，用户linux_groupA_admin可以管理（比如杀死任何一个作业或者改变任何作业的优先级）队列queueA，那么可以在mapred-queue-acls.xml中配置如下：

```
<configuration>
<property>
<name>mapred.queue.queueA.acl-submit-job</name>
<value>linux_userA linux_groupA</value>
</property>
<property>
<name>mapred.queue.queueA.acl-administer-jobs</name>
<value>linux_groupA_admin</value>
</property>
<!--配置其他队列.-->
</configuration>
```

2. 系统资源管理

Hadoop资源管理由调度器完成。管理员可在调度器中设置各个队列的资源容量、各个用户可用资源量等信息，而调度器则按照相应的资源约束对作业进行调度。考虑到系统中的队列信息是在mapred-site.xml中设置的，而队列资源分配信息在各个调度器的配置文件中设置，因此，这两个配置文件中的队列信息应保持一致是十分重要的。如果调度器中的某个队列在mapred-site.xml中没有设置，则意味着该队列中的资源无法得到使用。

通常而言，不同的调度器对资源管理的方式是不同的。接下来将介绍Capacity Scheduler和Fair Scheduler两个调度器的工作原理。

10.4 Capacity Scheduler实现

Capacity Scheduler^[1]是Yahoo! 开发的多用户调度器。它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用，而当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。总之，Capacity Scheduler主要有以下几个特点。

□容量保证：管理员可为每个队列设置资源最低保证和资源使用上限，而所有提交到该队列的作业共享这些资源。

□灵活性：如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的作业提交，则其他队列释放资源后会归还给该队列。相比于HOD调度器，这种资源灵活分配的方式可明显提高资源利用率。

□多重租赁：支持多用户共享集群和多作业同时运行。为防止单个作业、用户或者队列独占集群中的资源，管理员可为之增加多重约束（比如单个作业同时运行的任务数等）。

□支持资源密集型作业：当一个作业的单个任务需要的资源高于默认设置时，可同时为其分配多个slot，但需要注意的是，当前仅支持内存密集型作业。

□支持作业优先级：默认情况下，在每个队列中，空闲资源优先分配给最早提交的作业，但也可让其支持作业优先级，这样，优先级高的作业将优先获取资源（两个作业优先级相同时，再按照提交时间优先的原则分配资源）。需要注意的是，当前Capacity Scheduler还不支持资源抢占，也就是说，如果优先级高的作业提交时间晚于优先级低的作业，则高优先级作业需等待低优先级作业释放资源。

10.4.1 Capacity Scheduler功能介绍

Capacity Scheduler是一个多用户调度器。它设计了多层级别的资源限制条件以更好地让多用户共享一个Hadoop集群，比如队列资源限制、用户资源限制、用户作业数目限制等。为了能够更详尽地了解Capacity Scheduler的功能，我们从它的配置文件讲起。Capacity Scheduler有自己的配置文件，即存放在conf目录下的capacity-scheduler.xml。

在Capacity Scheduler的配置文件中，队列queueX的参数Y的配置名称为mapred.capacity-scheduler.queue.queueX.Y，为了简单起见，我们记为Y，则每个队列可以配置的参数如下。

□capacity：队列的资源容量（百分比）。当系统非常繁忙时，应保证每个队列的容量得到满足，而如果每个队列作业较少，可将剩余资源共享给其他队列。注意，所有队列的容量之和应小于100。

□maximum-capacity：队列的资源使用上限（百分比）。由于存在资源共享，因此一个队列使用的资源量可能超过其容量，而最多使用资源量可通过该参数限制。

□supports-priority：是否支持作业优先级。默认情况下，每个队列内部，提交时间早的作业优先获得资源，而如果支持优先级，则优先级高的作业优先获得资源，如果两个作业优先级相同，则再进一步考虑提交时间。

□minimum-user-limit-percent：每个用户最低资源保障（百分比）。任何时刻，一个队列中每个用户可使用的资源量均有一定的限制。当一个队列中同时运行多个用户的作业时，每个用户的可使用资源量在一个最小值和最大值之间浮动，其中，最小值取决于正在运行的作业数目，而最大值则由minimum-user-limit-percent决定。比如，假设minimum-user-limit-percent为25。当两个用户向该队列提交作业时，每个用户可使用资源量不能超过50%；如果三个用户提交作业，则每个用户可使用资源量不能超过33%；如果四个或者更多用户提交作业，则每个用户可使用资源量不能超过25%。

□ **user-limit-factor**: 每个用户最多可使用的资源量（百分比）。比如，假设该值为30，则任何时刻，每个用户使用的资源量不能超过该队列容量的30%。

□ **maximum-initialized-active-tasks**: 队列中同时被初始化的任务数目上限。通过设置该参数可防止因过多的任务被初始化而占用大量内存。

□ **maximum-initialized-active-tasks-per-user**: 每个用户可同时被初始化的任务数目上限。

□ **init-accept-jobs-factor**: 用于计算队列中可同时被初始化的作业数目上限，即为 $(\text{init-accept-jobs-factor}) \times (\text{maximum-system-jobs}) \times \text{capacity}/100$

其中，**maximum-system-jobs**为系统中最多可被初始化的作业数目。

一个配置文件实例如下：

```
<configuration>
<property>
<name>mapred.capacity-scheduler.maximum-system-jobs</name>
<value>3000</value>
<description>系统中最多可被初始化的作业数目</description>
</property>
<property>
<name>mapred.capacity-scheduler.maximum-system-jobs</name>
<value>3000</value>
<description>Hadoop集群中最多同时被初始化的作业</description>
</property>
<property>
<name>mapred.capacity-scheduler.queue.myQueue.capacity</name>
<value>30</value>
<description>default队列的可用资源（百分比）</description>
</property>
<!--配置myQueue队列.-->
<property>
<name>mapred.capacity-scheduler.queue.myQueue.maximum-capacity</name>
<value>40</value>
<description>default队列的资源使用上限（百分比）</description>
</property>
<property>
<name>mapred.capacity-scheduler.queue.myQueue.supports-priority</name>
<value>false</value>
<description>是否考虑作业优先级</description>
</property>
<property>
<name>mapred.capacity-scheduler.queue.myQueue.minimum-user-limit-percent</name>
<value>100</value>
<description>每个用户最低资源保障（百分比）</description>
</property>
<property>
<name>mapred.capacity-scheduler.queue.myQueue.user-limit-factor</name>
<value>1</value>
<description>每个用户最多可使用的资源占队列总资源的比例</description>
</property>
<property>
<name>mapred.capacity-scheduler.queue.myQueue.maximum-initialized-active-tasks</name>
<value>200000</value>
<description>default队列可同时被初始化的任务数目</description>
</property>
<property>
<name>mapred.capacity-scheduler.queue.myQueue.maximum-initialized-active-tasks-per-user</name>
<value>100000</value>
<description>default队列中每个用户可同时被初始化的任务数目</description>
</property>
<property>
<name>mapred.capacity-scheduler.queue.myQueue.init-accept-jobs-factor</name>
<value>10</value>
<description>default队列中可同时被初始化的作业数目，即该值与（maximum-system-jobs*
queue-capacity）的乘积</description>
</property>
<!--配置myQueue队列.-->
</configuration>
```

从上面这些参数可以看出，**Capacity Scheduler**将整个系统资源分成若干个队列，且每个队列有较为严格的资源使用限制，包括每个队列的资源容量限制、每个用户的资源量限制等。通过这些限制，**Capacity Scheduler**将整个Hadoop集群逻辑上划分成若干个拥有相对独立资源的子集群，而由于这些子集群实际上共用大集群中的资源，因此可以共享资源，相对于HOD而言，提高了

资源利用率且降低了运维成本。

[1] http://hadoop.apache.org/docs/stable/capacity_scheduler.html

1.Capacity Scheduler整体架构

第5章中已经讲了作业从提交到调度所经历的几个步骤。对于Capacity Scheduler而言，JobTracker启动时，会自动加载调度器类org.apache.hadoop.mapred.CapacityTaskScheduler（管理员需在参数mapred.jobtracker.taskScheduler中指定），而CapacityTaskScheduler启动时会加载自己的配置文件capacity-scheduler.xml，并向JobTracker注册监听器以随时获取作业变动信息。待调度器启动完后，用户可以提交作业。如图10-3所示，一个作业从提交到开始调度所经历步骤大致如下：

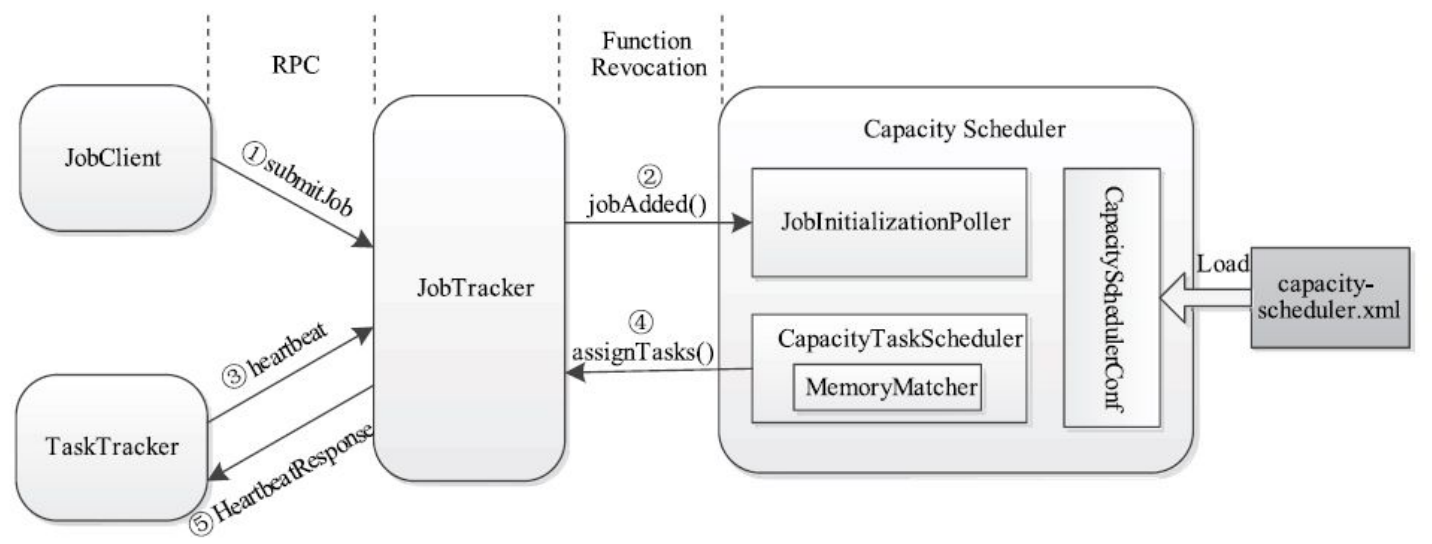


图 10-3 作业从提交到开始调度的整个过程

- 步骤1 用户通过Shell命令提交作业后，JobClient会将作业提交到JobTracker端。
- 步骤2 JobTracker通过监听器机制，将新提交的作业同步给Capacity Scheduler中的监听器JobQueuesManager；JobQueuesManager收到新作业后将作业添加到等待队列中，由JobInitializationPoller线程按照一定的策略对作业进行初始化。
- 步骤3 某一时刻，一个TaskTracker向JobTracker汇报心跳，且它心跳信息中要求JobTracker为其分配新的任务。
- 步骤4 JobTracker检测到TaskTracker可以接收新的任务后，调用CapacityTaskScheduler.assignTasks()函数为其分配任务。
- 步骤5 JobTracker将分配到的新任务返回给对应的TaskTracker。

接下来将重点介绍作业初始化和作业调度相关实现。

2.作业初始化

一个作业经初始化后才能够进一步得到调度器的调度而获取计算资源，因此，作业初始化是作业开始获取资源的前提。一个初始化的作业会占用JobTracker内存，因此需防止大量不能立刻得到调度的作业被初始化而造成内存浪费。Capacity Scheduler通过优先初始化那些最可能被调度器调度的作业和限制用户初始化作业数目来限制内存使用量。

由于作业经初始化后才能得到调度，因此，如果任务初始化的速度慢于被调度速度，则可能会产生空闲资源等待任务的现象。为了避免该问题，Capacity Scheduler总会过量初始化一些任务，从而让一部分任务处于等待资源的状态。

Capacity Scheduler中作业初始化由线程JobInitializationPoller完成。该线程由若干个（可通过参数mapred.capacity-scheduler.init-worker-threads指定，默认是5）工作线程JobInitializationThread组成，每个工作线程负责一个或者多个队列的作业初始化工作。作业初始化流程如下：

- 步骤1 用户将作业提交到JobTracker端后，JobTracker会向所有注册的监听器广播该作业信息；Capacity Scheduler中的监听器JobQueuesManager收到新作业添加的信息后，检查是否能够满足以下三个约束，如果不满足，则提示作业初始化失败，否则将该作业添加

到对应队列的等待作业列表中：

□ 该作业的任务数目不超过`maximum-initialized-active-tasks-per-user`。

□ 队列中等待初始化和已经初始化的作业数目不超过 $(\text{init-accept-jobs-factor}) \times (\text{maximum-system-jobs}) \times \text{capacity}/100$ 。

□ 该用户等待初始化和已经初始化的作业数目不超过 $[(\text{maximum-system-jobs}) \times \text{capacity}/100.0 \times (\text{minimum-user-limit-percent}) 100.0] \times (\text{init-accept-jobs-factor})$ 。

步骤2 在每个队列中，按照以下策略对未初始化的作业进行排序：如果支持作业优先级（`supports-priority`为`true`），则按照FIFO策略（先按照作业优先级排序，再按照到达时间排序）排序，否则，按照作业到达时间排序。每个工作线程每隔一段时间（可通过参数`mapred.capacity-scheduler.init-poll-interval`设定，默认是3 000毫秒）遍历其对应的作业队列，并选出满足以下几个条件的作业：

□ 队列已初始化作业数目（正运行的作业数目与已初始化但未运行作业数目之和）不超过 $(\text{maximum-system-jobs}) \times \text{capacity}/100.0$ 。

□ 队列中已初始化任务数目不超过`maximum-initialized-active-tasks`。

□ 该用户已经初始化作业数目不超过 $[(\text{maximum-system-jobs}) \times \text{capacity}/100.0 \times (\text{minimum-user-limit-percent}) / 100.0]$ 。

□ 该用户已经初始化的任务数目不超过`maximum-initialized-active-tasks-per-user`。

步骤3 调用`JobTracker.initJob()`函数对筛选出来的作业进行初始化。

3.任务调度

每个`TaskTracker`周期性向`JobTracker`发送心跳汇报任务进度和资源使用情况，并在出现空闲资源时请求分配新任务。当需要为某个`TaskTracker`分配任务时，`JobTracker`会调用调度器的`assignTasks`函数为其返回一个待运行的任务列表。对于`Capacity Scheduler`而言，该`assignTasks`函数由类`CapacityTaskScheduler`实现。其主要工作流程如图10-4所示，主要分为三个步骤：

步骤1 更新队列资源使用量。在选择任务之前，需要更新各个队列的资源使用信息，以便根据最新的信息进行调度。更新的信息包括队列资源容量、资源使用上限^[1]、正在运行的任务和已经使用的资源量等。



图 10-4 Capacity Scheduler中任务分配过程

步骤2 选择Map Task。正如第6章所述，Hadoop调度器通常采用三级调度策略，即依次选择一个队列、该队列中的一个作业和该作业

中的一个任务，Capacity Scheduler也是如此。下面分别介绍Capacity Scheduler采用的调度策略。

□ 选择队列：Capacity Scheduler总是优先将资源分配给资源使用率最低的队列，即numSlotsOccupied/capacity最小的队列，其中numSlotsOccupied表示队列当前已经使用的slot数目，capacity为队列的资源容量。

□ 选择作业：在队列内部，待调度作业排序策略与待初始化作业排序策略一样，即如果支持作业优先级（supports-priority为true），则按照FIFO策略排序，否则，按照作业到达时间排序。当选择任务时，调度器会依次遍历排好序的作业，并检查当前TaskTracker剩余资源是否足以运行当前作业的一个任务（注意，一个任务可能同时需要多个slot），如果满足，则从该作业中选择一个任务添加到已分配任务列表中。任务分配过程如图10-5所示。

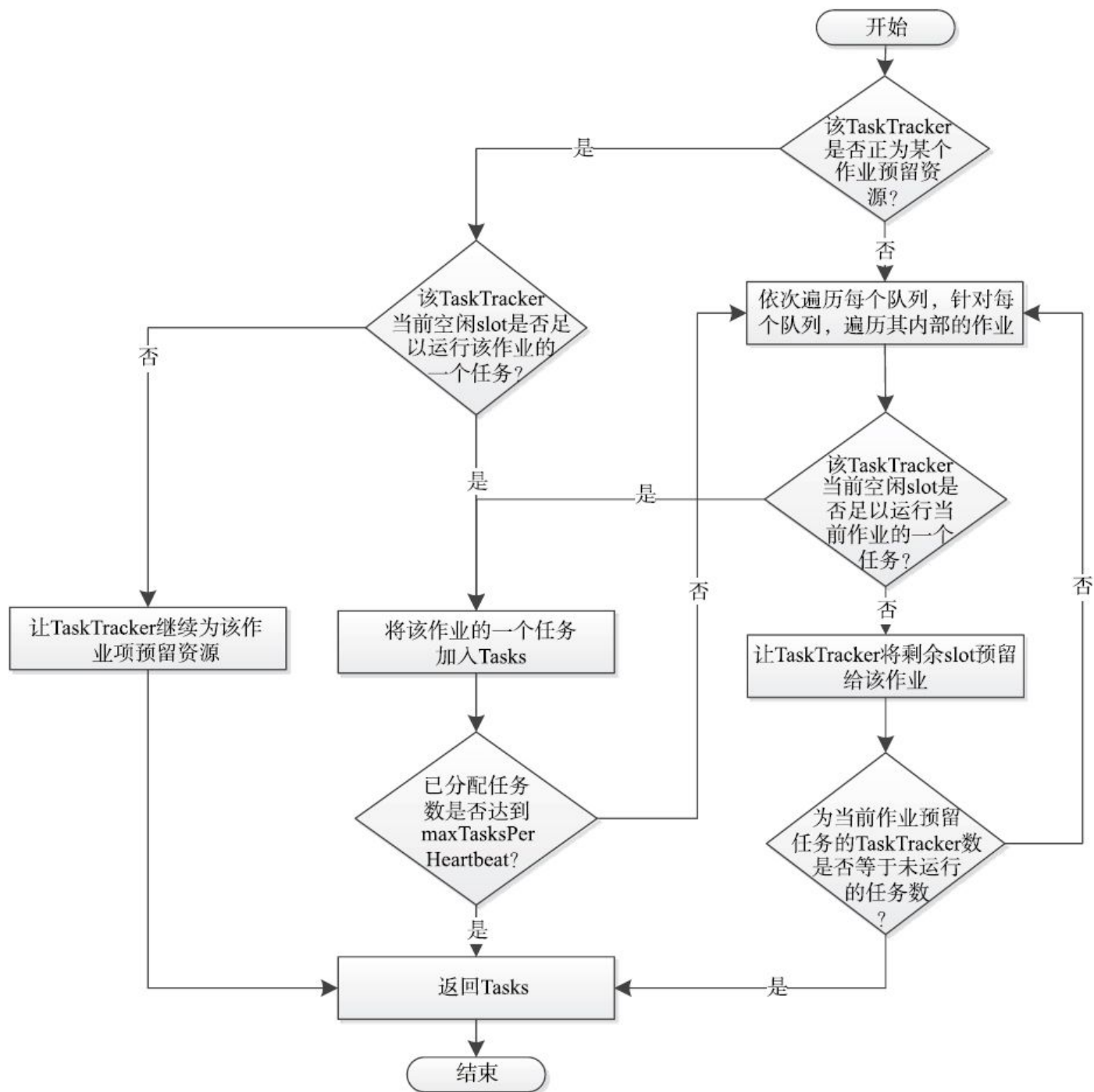


图 10-5 Capacity Scheduler任务分配流程图

Capacity Scheduler调度过程用到了以下几个机制。

机制1：大内存任务调度。

Capacity Scheduler提供了对大内存任务的调度机制。默认情况下，Hadoop假设所有任务是同质的，任何一个任务只能使用一个slot，考虑到一个slot代表的内存是一定的，因此这并没有考虑那些内存密集型的任务。为解决该问题，Capacity Scheduler可根据任务的内存需求量为其分配一个或者多个slot。如果当前TaskTracker空闲slot数目小于作业的单个任务的需求量，调度器会让TaskTracker为该作业预留^[2]当前空闲的slot，直到累计预留的slot数目满足当前作业的单个任务需求，此时，才会真正地将该任务分配给TaskTracker执行。

默认情况下，大内存任务调度机制是关闭的，只有当管理员配置了mapred.cluster.map.memory.mb、mapred.cluster.reduce.memory.mb、mapred.cluster.max.map.memory.mb、mapred.cluster.max.reduce.memory.mb四个参数后，才会支持大内存任务调度，此时，调度器会按照以下公式计算每个Map Task需要的slot数（Reduce Task计算方法类似）：

$$\lceil \frac{\text{\$}\{\text{mapred.job.map.memory.mb}\}}{\text{\$}\{\text{mapred.cluster.map.memory.mb}\}} \rceil$$

机制2：通过任务延迟调度以提高数据本地性。

第6章提到，当任务的输入数据与分配到的slot位于同一个节点或者机架时，称该任务具有数据本地性。数据本地性包含三个级别，分别是node-local（输入数据和空闲slot位于同一个节点）、rack-local（输入数据和空闲slot位于同一个机架）和off-switch（输入数据和空闲slot位于不同机架）。由于为空闲slot选择具有本地性的任务可避免通过网络远程读取数据进而提高数据读取效率，所以Hadoop会优先选择node-local的任务，然后是rack-local，最后是off-switch。

为了提高任务的数据本地性，Capacity Scheduler采用了作业延迟调度的策略：当选中一个作业后，如果在该作业中未找到满足数据本地性的任务，则调度器会让该作业跳过一定数目的调度机会，直到找到一个满足本地性（node-local或rack-local）的任务或者达到跳过次数上限（requiredSlots×localityWaitFactor），其中，localityWaitFactor可通过参数mapreduce.job.locality.wait.factor配置，默认情况下，计算方法如下：

$$\text{localityWaitFactor} = \min\{\text{jobNodes}/\text{clusterNodes}, 1\}$$

其中，jobNodes表示该作业输入数据所在的节点总数；clusterNodes表示整个集群中节点总数。

requiredSlots计算方法如下：

$$\text{requiredSlots} = \min\{(\text{numMapTasks} - \text{finishedMapTask}), \text{numTaskTrackers}\}$$

其中，numMapTasks、finishedMapTasks分别表示该作业总的Map Task数目和已经运行完成的Map Task数目；numTaskTrackers表示整个集群中的TaskTracker数目（注意，由于一个节点上可能用于多个TaskTracker，因此numTaskTrackers与clusterNodes可能不相等）。

机制3：批量任务分配。

为了加快任务分配速度，Capacity Scheduler支持批量任务分配，管理员可通过参数mapred.capacity-scheduler.maximum-tasks-per-heartbeat（默认是Short.MAX_VALUE）指定一次性为一个TaskTracker分配的最多任务数。需要注意的是，该机制倾向于将任务分配给优先发送心跳的TaskTracker，也就是说，当系统slot数目大于任务需要的数目时，会使得任务集中运行在少数几个节点上，且同一个作业的任务也可能会集中分配到几个节点上，这不利于负载均衡。

步骤3：选择Reduce Task。

相比于Map Task, Reduce Task选择机制就简单多了。它仅采用了大内存任务调度策略，至于其他策略，如任务延迟调度（Reduce Task没有数据本地性）和批量任务分配等，不再采用。调度器只要找到一个合适的Reduce Task便可以返回。

[1] 队列的资源容量和资源使用上限是在配置文件中配置的百分比。在一个运行的Hadoop集群中，节点的数目是不断变化的，因此，通过该百分比求出来的资源量也是变化的。

[2] 预留：暂时占下slot，尽管没有实际使用，但可防止被其他任务占用。

10.4.3 多层队列调度

在Hadoop 1.0中，队列以平级结构组织在一起，且每个队列不能再进一步划分。但在实际应用中，每个队列可能代表一个部门，该部门可能又进一步划分成若干个子部门或者将自己的资源按照应用类型划分到不同队列中，最终形成一个树形组织结构。一个典型的例子如图10-6所示。

为了支持这种多层队列组织方式，在Hadoop 2.0中，Capacity Scheduler在现有实现基础上添加了对多层队列的支持，主要特性如下：

- 整个组织结构由中间队列和叶子队列组成，其中，中间队列包含若干子队列，而叶子队列没有再分解的队列。
- 任何队列可划分成若干子队列，但子队列容量之和不能超过父队列总容量。
- 用户作业只能将作业提交到某个叶子队列中。
- 当某个队列出现空闲资源时，优先共享给同父亲的其他子队列。以图10-6为例，当队列C11中有剩余资源时，首先共享给C12，其次是C2，最后才是A1，A2和B1。
- 进行任务调度时，仅考虑叶子队列，且采用的调度机制与现有的调度机制一致。

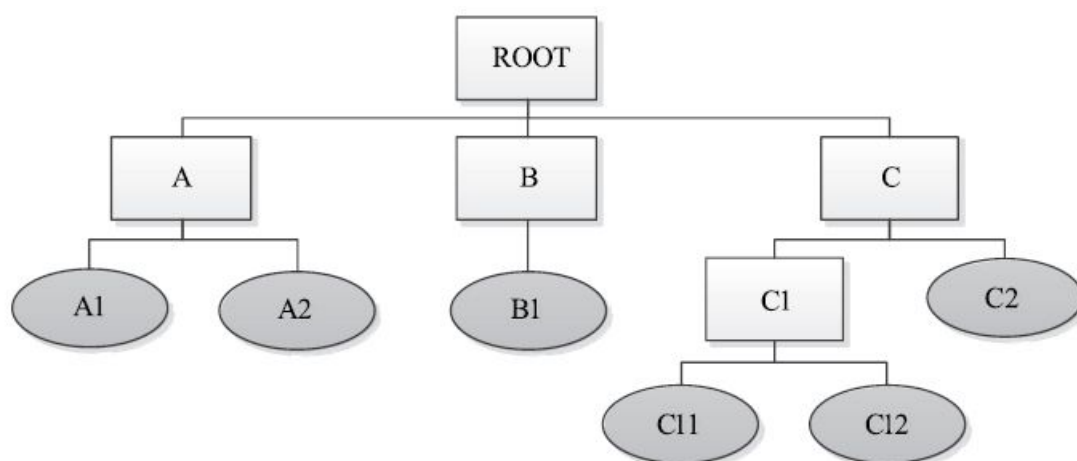


图 10-6 Capacity Scheduler多层队列结构图

10.5 Fair Scheduler实现

Fair Scheduler^[1]是Facebook开发的多用户调度器。与Capacity Scheduler类似，它以资源池（与队列一个概念）为单位划分资源，每个资源池可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用；当一个资源池的资源有剩余时，可暂时将剩余资源共享给其他资源池。当然，Fair Scheduler也存在很多与Capacity Scheduler不同之处，主要体现在以下几个方面。

❑资源公平共享：在每个资源池中，Fair Scheduler可选择按照FIFO或者Fair策略为作业分配资源，其中Fair策略是一种基于最大最小公平算法^[2]实现的资源多路复用方式，默认情况下，每个队列内部采用该方式分配资源。这意味着，如果一个队列中有两个作业同时运行，则每个作业可得到1/2的资源；如果三个作业同时运行，则每个作业可得到1/3的资源。

❑支持资源抢占：当某个资源池中有剩余资源时，调度器会将这些资源共享给其他资源池；而当该资源池中有新的作业提交时，调度器要为其回收资源。为了尽可能降低不必要的计算浪费，调度器采用了先等待再强制回收的策略，即如果等待一段时间尚有未归还的资源，则会进行资源抢占：从那些超额使用资源的队列中杀死一部分任务，进而释放资源。

❑负载均衡：Fair Scheduler提供了一个基于任务数目的负载均衡机制。该机制尽可能将系统中的任务均匀分配到各个节点上。此外，用户也可以根据自己的需要设计负载均衡机制。

❑任务延时调度：Fair Scheduler提供了一种基于延时等待的调度机制以提高任务的数据本地性。该机制通过暂时减少个别作业的资源量而提高系统整体吞吐量。

❑降低小作业调度延迟：由于采用了最大最小公平算法，小作业可以优化获取资源并运行完成。

10.5.1 Fair Scheduler功能介绍

与Capacity Scheduler类似，Fair Scheduler也是一个多用户调度器，它同样添加了多层级别的资源限制条件以更好地让多用户共享一个Hadoop集群，比如队列资源限制、用户作业数目限制等。然而，由于Fair Scheduler增加了很多新的特性，因此它的配置选项更多。为了能够更详尽地了解Fair Scheduler的功能，我们从它的配置文件讲起。Fair Scheduler的配置选项包括两部分，其中一部分在mapred-site.xml中，另外一部分在自己的配置文件中，默认情况下为存放在conf目录下的fair-scheduler.xml。

1.配置文件mapred-site.xml

启用Fair Scheduler时，可在配置文件mapred-site.xml中增加以下几个配置选项（其中，mapred.jobtracker.taskScheduler是必填的，其他自选）。

❑mapred.jobtracker.taskScheduler：采用的调度器所在的类，即为org.apache.hadoop.mapred.FairScheduler。

❑mapred.fairscheduler.poolnameproperty：资源池命名方式，包含以下三种命名方式。

○user.name：默认值，一个UNIX用户对应一个资源池。

○group.name：一个UNIX用户组对应一个资源池。

○mapred.job.queue.name：一个队列对应一个资源池。如果设置为该值，则与Capacity Scheduler一样。

❑mapred.fairscheduler.allocation.file：Fair Scheduler配置文件所在位置，默认是\$HADOOP_HOME/conf/fair-scheduler.xml。

❑mapred.fairscheduler.preemption：是否支持资源抢占，默认为false。

❑ `mapred. fairscheduler.preemption.only.log`: 是否只打印资源抢占日志，并不真正进行资源抢占。打开该选项可用于调试。

❑ `mapred. fairscheduler.assignmultiple`: 是否在一次心跳中同时分配Map Task和ReduceTask，默认为true。

❑ `mapred. fairscheduler.assignmultiple.maps`: 一次心跳最多分配的Map Task数目，默认是-1，表示不限制。

❑ `mapred. fairscheduler.assignmultiple.reduces`: 一次心跳最多分配的Reduce Task数目，默认是-1，表示不限制。

❑ `mapred. fairscheduler.sizebasedweight`: 是否按作业大小调整作业权重。将该参数置为true后，调度器会根据作业长度（任务数目）调整作业权重，以让长作业获取更多资源，默认是false。

❑ `mapred. fairscheduler.locality.delay.node`: 为了等待一个满足node-local的slot，作业可最长等待时间。

❑ `mapred. fairscheduler.locality.delay.rack`: 为了等待一个满足rack-local的slot，可最长等待时间。

❑ `mapred. fairscheduler.loadmanager`: 可插拔负载均衡器。用户可通过继承抽象类LoadManager实现一个负载均衡器，以决定每个TaskTracker上运行的Map Task和Reduce Task数目，默认实现是CapBasedLoadManager，它将集群中所有Task按照数量平均分配到各个TaskTracker上。

❑ `mapred. fairscheduler.taskselector`: 可插拔任务选择器。用户可通过继承TaskSelector抽象类实现一个任务选择器，以决定对于给定一个TaskTracker，为其选择作业中的哪个任务。具体实现时可考虑数据本地性，推测执行等机制。默认实现是DefaultTaskSelector，它使用了JobInProgress中提供的算法，具体可参考第6章。

❑ `mapred. fairscheduler.weightadjuster`: 可插拔权重调整器。用户可通过实现WeightAdjuster接口编写一个权重调整器，以动态调整运行作业的权重。

2. 配置文件fair-scheduler.xml

fair-scheduler.xml是Fair Scheduler的配置文件，管理员可为每个pool添加一些资源约束以限制资源使用。对于每个pool，用户可配置以下几个选项。

❑ `minMaps`: 最少保证的Map slot数目，即最小资源量。

❑ `maxMaps`: 最多可以使用的Map slot数目。

❑ `minReduces`: 最少保证的Reduce slot数目，即最小资源量。

❑ `maxReduces`: 最多可以使用的Reduce slot数目。

❑ `maxRunningJobs`: 最多同时运行的作业数目。通过限制该数目，可防止超量MapTask同时运行时产生的中间输出结果撑爆磁盘。

❑ `minSharePreemptionTimeout`: 最小共享量抢占时间。如果一个资源池在该时间内使用的资源量一直低于最小资源量，则开始抢占资源。

❑ `schedulingMode`: 队列采用的调度模式，可以是FIFO或者Fair。

管理员也可为单个用户添加maxRunningJobs属性限制其最多同时运行的作业数目。此外，管理员也可通过以下参数设置以上属性的默认值。

❑ `poolMaxJobsDefault`: 资源池的`maxRunningJobs`属性的默认值。

❑ `userMaxJobsDefault`: 用户的`maxRunningJobs`属性的默认值。

❑ `defaultMinSharePreemptionTimeout`: 资源池的`minSharePreemptionTimeout`属性的默认值。

❑ `defaultPoolSchedulingMode`: 资源池的`schedulingMode`属性的默认值。

❑ `fairSharePreemptionTimeout`: 公平共享量抢占时间。如果一个资源池在该时间内使用资源量一直低于公平共享量的一半，则开始抢占资源。

❑ `defaultPoolSchedulingMode`: Pool的`schedulingMode`属性的默认值。

【实例】假设要为一个Hadoop集群增加三个资源池`poolA`、`poolB`和`default`，且规定普通用户最多可同时运行40个作业，但用户`userA`最多可同时运行400个作业，那么可在`fair-scheduler.xml`中进行如下配置：

```
<allocations>
  <pool name="poolA">
    <minMaps>100</minMaps>
    <maxMaps>150</maxMaps>
    <minReduces>50</minReduces>
    <maxReduces>100</maxReduces>
    <maxRunningJobs>200</maxRunningJobs>
    <minSharePreemptionTimeout>300</minSharePreemptionTimeout>
    <weight>1.0</weight>
  </pool>
  <pool name="poolB">
    <minMaps>80</minMaps>
    <maxMaps>80</maxMaps>
    <minReduces>50</minReduces>
    <maxReduces>50</maxReduces>
    <maxRunningJobs>30</maxRunningJobs>
    <minSharePreemptionTimeout>500</minSharePreemptionTimeout>
    <weight>1.0</weight>
  </pool>
  <pool name="default">
    <minMaps>0</minMaps>
    <maxMaps>10</maxMaps>
    <minReduces>0</minReduces>
    <maxReduces>10</maxReduces>
    <maxRunningJobs>50</maxRunningJobs>
    <minSharePreemptionTimeout>500</minSharePreemptionTimeout>
    <weight>1.0</weight>
  </pool>
  <user name="userA">
    <maxRunningJobs>400</maxRunningJobs>
  </user>
  <userMaxJobsDefault>40</userMaxJobsDefault>
  <fairSharePreemptionTimeout>6000</fairSharePreemptionTimeout>
</allocations>
```

[1] http://hadoop.apache.org/docs/stable/fair_scheduler.html

[2] Max-Min Fairness (Wikipedia) : http://en.wikipedia.org/wiki/Max-min_fairness

1.Fair Scheduler基本设计思想

Fair Scheduler核心设计思想是基于资源池的最小资源量和公平共享量进行任务调度。其中，最小资源量是管理员配置的，而公平共享量是根据队列或作业权重计算得到的。资源分配具体过程如下：

- 步骤1 根据最小资源量将所有系统中所有slot分配给各个资源池。如果某个资源池实际需要的资源量小于它的最小资源量，则只需将实际资源需求量分配给它即可。
- 步骤2 根据资源池的权重将剩余的资源分配给各个资源池。
- 步骤3 在各个资源池中，按照作业权重将资源分配给各个作业，最终每个作业可以分到的资源量即为作业的公平共享量。其中，作业权重是由作业优先级转换而来的，它们的映射关系如表10-1所示。

表 10-1 作业优先级与作业权重映射关系

作业优先级	作业权重
VERY_HIGH	4.0
HIGH	2.0
NORMAL	1.0
LOW	0.5
VERY_LOW	0.25

用户也可以通过打开`mapred.fairscheduler.sizebasedweigh`参数以根据作业长度调整权重或者编写权重调整器动态调整作业权重。

【实例】如图10-7所示，假设一个Hadoop集群中共有100有slot（为了简单，不区分Map或者Reduce slot）和四个资源池（依次为P1、P2、P3和P4），它们的最小资源量依次为：25、19、26和28。如图10-7 a所示，在某一时刻，四个资源池实际需要的资源量（与未运行的任务数目相关）依次为20、26、37和30，则资源分配过程如下：

- 步骤1 根据最小资源量将资源分配各个资源池。对于资源池P1而言，由于它实际需要的资源量少于其最小资源量，因此只需将它实际需要的资源分配给它即可，如图10-7b和图10-7c所示。经过这一轮分配，四个资源池获得的slot数目依次为：20、19、26和28。
- 步骤2 经过第一轮分配后，尚剩余7个slot，此时需按照权重将剩余资源分配给尚需资源的资源池P2，P3和P4。不妨假设这三个资源池的权重依次为2.0、3.0和2.0，则它们额外分得的slot数目依次为2、3和2。这样，如图10-7 d所示，三个资源池最终获得的资源总量依次为：21、29和30。
- 步骤3 在各个资源池内部，按照作业的权重将资源分配给各个作业。以P4为例，假设P4中有三个作业，优先级依次为VERY_HIGH, NORMAL和NORMAL，则它们可能获得的slot数目依次为： $\frac{4}{4+1+1} \times 30=20$ ， $\frac{1}{4+1+1} \times 30=5$ 和 $\frac{1}{4+1+1} \times 30=5$ ，这三个值即为对应作业的公平共享量。

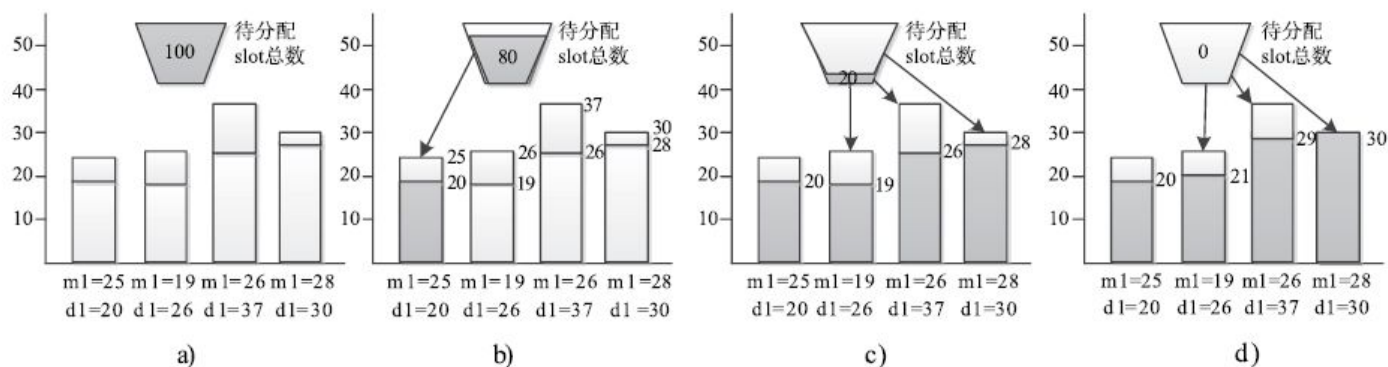


图 10-7 Fair Scheduler中的任务分配过程

注意 公平共享量只是理论上的资源分配量（理想值），在实际资源分配时，调度器应尽量将与公平共享量相等的资源分配给作业。

2.Fair Scheduler实现

Fair Scheduler内部组织结构如图10-8所示。涉及的模块有：配置文件加载模块、作业监听模块、状态更新模块和调度模块。下面分别介绍这几个模块。

□ 配置文件加载模块：由类PoolManager完成，负责将配置文件fair-scheduler.xml中的信息加载到内存中。

□ 作业监听模块：Fair Scheduler启动时会向JobTracker注册作业监听器JobListener，以便能够随时获取作业变化信息。

□ 状态更新模块：由线程UpdateThread完成，该线程每隔mapred.fairscheduler.update.interval（默认是500毫秒）时间更新一次队列和作业的信息，以便将最新的信息提供给调度模块进行任务调度。

□ 调度模块：当某个TaskTracker通过心跳请求任务时，该模块根据最新的队列和作业的信息为该TaskTracker选择一个或多个任务。

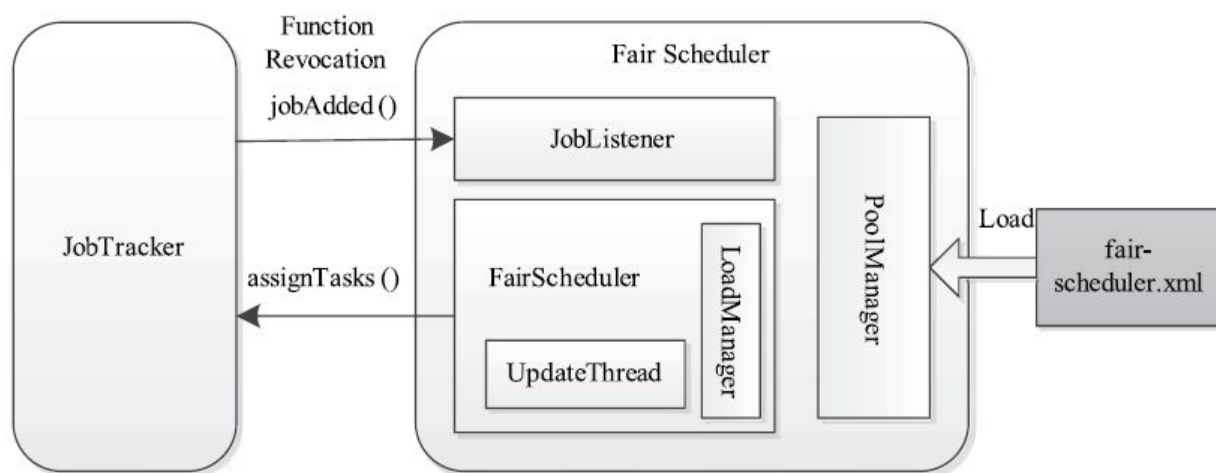


图 10-8 Fair Scheduler内部组织结构

在不同的Hadoop版本中，Fair Scheduler调度算法实现方式不同。这里介绍两个版本的实现：0.20.X版本和0.21.X/0.22.X/1.X版本。

(1) 0.20.X版本

前面提到了作业公平共享量的计算方法，而调度器的任务就是将与公平共享量相等的资源分配给作业。在实际的Hadoop集群中，由于资源使用情况是动态变化的，且任务运行的时间长短不一，因此时刻保证每个作业实际分到的资源量与公平共享量一致是不可能的。为此，0.20.X版本采用了基于缺额的调度策略。该策略采用了贪心算法以保证尽可能公平地将资源分配给各个作业。

缺额（jobDeficit）是作业的公平共享量与实际分配到的资源量之间的差值。它反映了资源分配过程中产生的“理想与现实的差距”。调度器在实际资源分配时，应保证所有作业的缺额尽可能小。缺额的基本计算公式为：

$$\text{jobDeficit} = \text{jobDeficit} + (\text{jobFairShare} - \text{runningTasks}) \times \text{timeDelta}$$

其中，`jobFairShare`为作业的公平共享量，`runningTasks`为作业正在运行的任务数目（对应实际分配到的资源量），`timeDelta`为缺额更新时间间隔。

从上面公式可以看出，作业缺额是随着时间积累的。在进行资源分配时，调度器总是优先将空闲资源分配给当前缺额最大的作业。如果在一段时间内一个作业一直没有获得资源，则它的缺额会越来越大，最终缺额变得最大，从而可以获得资源。这种基于缺额的调度机制并不能保证作业时时刻刻均能获得与其公平共享量对应的资源，但如果所有作业的运行时间足够长，则该机制能够保证每个作业实际平均分配到的资源量逼近它的公平共享量。

（2）0.21.X/0.22.X/1.X版本

在0.21.X/0.22.X/1.X版本中，同Capacity Scheduler一样，Fair Scheduler也采用了三级调度策略，即依次选择一个资源池、该资源池中的一个作业和该作业中的一个任务，但具体采用的策略稍有不同。

选择队列

Fair Scheduler选择队列时，在不同的条件下采用不同的策略，具体如下：

□ 当存在资源使用量小于最小资源量的资源池时，优先选择资源使用率最低的资源池，即`runningTasks/minShare`最小的资源池，其中`runningTasks`是资源池当前正在运行的Task数目（也就是正在使用的slot数目），`minShare`为资源池的最小资源量。

□ 否则，选择任务权重比最小的资源池，其中资源池的任务权重比（`tasksToWeightRatio`）定义如下：

$$\text{tasksToWeightRatio} = \text{runningTasks} / \text{poolWeight}$$

其中，`runningTasks`为资源池中正在运行的任务数目；`poolWeight`是管理员配置的资源池权重。

选择作业

选定一个资源池后，Fair Scheduler总是优先将资源分配给资源池中任务权重比最小的作业，其中作业的任务权重比的计算方法与资源池的一致，即为该作业正在运行的任务数目与作业权重的比值。但需要注意的是，作业权重比是由作业优先级转换而来的。此外，Fair Scheduler为管理员提供了另外两种改变作业的权重的方法：

□ 将参数`mapred.fairscheduler.sizebasedweight`置为`true`，则计算作业权重时会考虑作业长度，具体计算方法如下：

$$\text{jobWeight} = \text{jobWeightByPriority} \times \log_2(\text{runnableTasks})$$

其中，`jobWeightByPriority`是通过优先级转化来的权重；`runnableTasks`是作业正在运行和尚未运行的任务之和。

□ 通过实现WeightAdjuster接口，编写一个权重调整器，并通过参数`mapred.fairscheduler.weightadjuster`使之生效，此时，作业权重即为WeightAdjuster中方法`adjustWeight`的返回值。

选择任务

任务选择策略已在第6章介绍过了，Fair Scheduler在该策略基础上又添加了延时调度机制^[1]，具体见下一小节。

3.Fair Scheduler优化机制

机制1：延时调度。

第6章中已讲了Map Task的数据本地性问题。我们知道，提高Map Task的数据本地性可提高作业运行效率。为了提高数据本地性，Fair Scheduler采用了延时调度机制：当出现一个空闲slot时，如果选中的作业没有`node-local`或者`rack-local`的任务，则暂时把资源让给其他作业，

直到找到一个满足数据本地性的任务或者达到一个时间阈值，此时不得不为之选择一个非本地性的任务。

为了实现延时调度，Fair Scheduler为每个作业j维护三个变量：level、wait和skipped，分别表示最近一次调度时作业的本地性级别（0、1、2分别对应node-local、rack-local和off-switch）、已等待时间和是否延时调度，并依次初始化为：j.level=0、j.wait=0和j.skipped=false。此外，当不存在node-local任务时，为了尽可能选择一个本地性较好的任务，Fair Scheduler采用了双层延迟调度算法：为了找到一个node-local任务最长可等待W1或者进一步等待W2找一个rack-local任务。

总之，当JobTracker从TaskTracker上收到心跳后，Fair Scheduler按照以下算法选择Map Task:

```
function List<Task>assignTasks (TaskTracker tt)
taskList←null
for each job j in allJobs do
if j.skipped=true do
j.updateLocalityWaitTimes(); //更新等待时间
j.skipped=false
done
end for
while n.availableSlots()>0 then//如果可用slot数目大于0
sort jobs using hierarchical scheduling policy
for j in jobs do//遍历排序后的所有作业
//查找该作业中是否包含符合node-local的Map Task
if (t=j.obtainNewNodeLocalMapTask()) !=null then
j.wait←0, j.level←0
taskList.add(t);
break;
//查找该作业中是否包含符合rack-local的Map Task
else if (t=j.obtainNewNodeOrRackLocalMapTask()) !=null and
(j.level>=1 or j.wait>=W1) then
j.wait←0, j.level←1
taskList.add(t)
break;
else if j.level=2 or (j.level=1 and j.wait>=W2) or
(j.level=0 and j.wait>=W1+W2) then
j.wait←0, j.level←2
//依次查找该作业中符合node-local, rack-local, off-switch的Map Task
t=j.obtainNewMapTask()
taskList.add(t);
break;
else
j.skipped=true
end if
end for
end while
return taskList;
end function
```

机制2：负载均衡。

Fair Scheduler为用户提供了一个可扩展的负载均衡器：CapBasedLoadManager。它会将系统中所有任务按照数量平均分配到各个节点上

[2]。当然，用户也可通过继承抽象类LoadManager实现自己的负载均衡器。

机制3：资源抢占。

当一个资源池有剩余资源时，Fair Scheduler会将这些资源暂时共享给其他资源池；而一旦该资源池有新作业提交，调度器则为它回收资源。如果在一段时间后该资源池仍得不到本属于自己的资源，则调度器会通过杀死任务的方式抢占资源。Fair Scheduler同时采用了两种资源抢占方式：最小资源量抢占和公平共享量抢占。如果一个资源池的最小资源量在一定时间内得不到满足，则会从其他超额使用资源的资源池中抢占资源，这就是最小资源量抢占；而如果一定时间内一个资源池的公平共享量的一半得不到满足，则该资源池也会从其他资源池中抢占，这称为公平共享量抢占。

进行资源抢占时，调度器会选出超额使用资源的资源池，并从中找出启动时间最早的任务，再将其杀掉，进而释放资源。

[1] M. Zaharia, D.Borthakur, J.Sen Sarma, K.Elmeleegy, S.Shenker, and I.Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling”in Proc.of EuroSys.ACM, 2010, pp.265-278.

[2] Hadoop 1.0.X中存在一个Bug，这使得在批量调度模式下不能实现负载均衡。该Bug在1.1.0版本中已经修复，具体参考：<https://issues.apache.org/jira/browse/MAPREDUCE-2905>。

10.5.3 Fair Scheduler与Capacity Scheduler对比

表10-2从多个方面对比了这两个调度器的异同。通过这个表，读者能更好地理解Capacity Scheduler与Fair Scheduler的相同点和不同点。

表 10-2 Capacity Scheduler 与 Fair Scheduler 比较

	Capacity Scheduler	Fair Scheduler
目标	提供一种多用户共享 Hadoop 集群的方法，以提高资源利用率和降低集群管理成本	
设计思想	资源按比例分配给各个队列，并添加各种严格的限制以防止个别用户或者队列独占资源	基于最大最小公平算法将资源分配给各个资源池或者用户
是否支持动态加载配置文件	否	是
是否支持负载均衡	否	是
是否支持资源抢占	否	是
是否支持大内存作业	是，允许一个 Task 占用多个 slot	否，一个 Task 只能占用一个 slot
是否支持批量调度	是	是
本地性任务调度优化	基于跳过次数的延迟调度	基于时间的延迟调度
队列间资源分配方式	资源使用率低者优先	
队列内部资源分配方式	默认是先来先服务，但也支持优先级	默认是 Fair，也支持 FIFO

10.6 其他Hadoop调度器介绍

1.自适应调度器

自适应调度器（Adaptive Scheduler）^[1]是一种以用户期望运行时间为目标的调度器。该调度器根据每个作业会被分解成多个任务的事实，通过已经运行完成的任务的运行时间估算剩余任务的运行时间，进而使得该调度器能够根据作业的进度和剩余时间动态地为作业分配资源，以期望作业在规定时间内运行完成。

2.自学习调度器

自学习调度器（Learning Scheduler）^[2]是一种基于贝叶斯分类算法的资源感知调度器。与现有的调度器不同，它更适用于异构Hadoop集群。该调度器的创新之处是将贝叶斯分类算法应用到MapReduce调度器设计中。

该调度器选取了若干个作业特征（用向量表示）作为分类属性，主要有作业平均CPU利用率、平均网络利用率、平均磁盘I/O利用率和平均内存利用率等。这些属性值可通过一个离线系统获取。调度器通过用户标注好的一些作业可训练得到一个分类器，这样，当某个TaskTracker出现剩余资源时，会通过心跳向JobTracker请求新的任务，同时汇报所在节点的资源使用信息。调度器收到该信息后，会将所有作业的特征向量作为贝叶斯分类器的输入，判断出当前哪些作业可在该TaskTracker上运行（称为“good”作业），哪些不可以在该TaskTracker上运行（称为“bad”作业），最后通过一个效用函数从所有“good”作业中选出一个最合适的作业。

3.动态优先级调度器

在0.21.X/0.22.X版本中，Hadoop引入了一个新的调度器——动态优先级调度器（Dynamic Priority Scheduler）^[3]。该调度器允许用户动态调整自己获取的资源量以满足其服务质量要求。

该调度器试图把Hadoop集群看作一个提供商品的买卖市场，每个消费者有一定的预算购买自己需要的东西，且消费者需为购买某件商品竞标，其中，出价高的人可获得较多的商品，反之，出价少的人获得的商品也少。由于市场中商品价格是不断上下波动的，因此消费者可结合自己的需要调整自己的价位以买入更多或者更少的产品。对应到Hadoop集群中，slot是进行买卖的商品，Hadoop用户是消费者，每个用户分配有一定的预算，在任何一个阶段，可能有多个用户同时向Hadoop集群申请资源，其中出价高的用户获得的资源多，且申请资源的用户越多，单个slot的价位也就越高。

动态优先级调度器的核心思想是在一定的预算约束下，根据用户提供的消费率按比例分配资源。管理员可根据集群资源总量为每个用户分配一定的预算和一个时间单元的长度（通常为10s~1min），而用户可根据自己的需要动态调整自己的消费率，即每个时间单元内单个slot的价钱。在每个时间单元内，调度器按照以下步骤计算每个用户获得的资源量：

1) 计算所有用户的消费率之和 p 。

2) 对于每个用户 i ，分配 $\frac{s_i}{p} \times c$ ，其中， s_i 为用户 i 的消费率， c 为Hadoop集群中slot总数。

3) 对于每个用户 i ，从其预算中扣除 $s_i \times u_i$ ，其中 u_i 为用户正在使用的slot数目。

动态优先级调度器也可作为一个元调度器集成到其他调度器（比如FIFO, Fair Scheduler等）中，这样，每个队列或者用户的可用资源量直接由动态优先级调度器动态计算得到，而其他调度器只需负责分配资源即可。相比于其他调度器，动态优先级调度器允许用户根据需要（比如完成时间）动态调整资源，进而可以对作业运行质量进行精细的控制。

^[1] <https://issues.apache.org/jira/browse/MAPREDUCE-1380>

[2] <https://issues.apache.org/jira/browse/MAPREDUCE-1439>

[3] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In JSSPP'10: 15th Workshop on Job Scheduling Strategies for Parallel Processing, 2010

10.7 小结

本章介绍了几种常见的多用户作业调度器。相比于FIFO调度器，多用户调度器能够更好地满足不同应用程序的服务质量要求。

当前主要有两种多用户作业调度器的设计思路：第一种是在一个物理集群上虚拟多个Hadoop集群，这些集群各自拥有全套独立的Hadoop服务，比如JobTracker、TaskTracker等，典型的代表是HOD（Hadoop On Demand）调度器；另一种是扩展Hadoop调度器，使之支持多个队列多用户，典型的代表是Yahoo! 的Capacity Scheduler和Facebook的Fair Scheduler。本章分别对这两种调度器进行了介绍。

HOD调度器是一个在共享物理集群上管理若干个Hadoop集群的工具，它可以帮助用户在一个共享物理集群上快速搭建若干个独立的虚拟Hadoop集群。由于该调度器会产生多个独立的小集群，因此会增加集群运维成本和降低资源利用率。

为了克服HOD的缺点，Capacity Scheduler和Fair Scheduler出现了。它们通过扩展调度器功能，在不拆分集群的前提下，将集群中的资源和用户分成若干个队列，并为每个队列分配一定量的资源，同时添加各种限制以防止用户或者队列独占资源。由于这种方式能够保证只有一个Hadoop集群，因此可大大降低运维成本，同时很容易实现资源共享，进而可明显提高资源利用率。

第11章 Hadoop安全机制

第10章中介绍了常见的几种Hadoop多用户任务调度器，包括Fair Scheduler、Capacity Scheduler等。多用户任务调度器使得不同需求的用户可以共享一个Hadoop集群中的计算资源和存储资源，进而大大降低了运维成本且提高了系统资源利用率，但同时引出了一个亟需解决的问题——安全问题。由于Hadoop缺乏安全机制，当大量用户共享一个Hadoop集群时，可能会带来各种安全隐患，比如普通用户访问机密数据，用户杀死他人的作业等。为了更好地管理Hadoop集群，从1.0版本开始，Hadoop引入了安全机制。

本章将从Hadoop安全设计动机出发，依次介绍Hadoop RPC、HDFS和MapReduce中的安全机制设计方法。

11.1 Hadoop安全机制概述

由于所有的Hadoop集群都部署在有防火墙保护的局域网中且只允许公司内部人员访问，因此为Hadoop添加安全机制的动机并不像传统的安全概念那样为了防御外部黑客的攻击，而是为了更好地让多用户在共享Hadoop集群环境下安全高效地使用集群资源。

11.1.1 Hadoop面临的安全问题

在1.0版本之前，Hadoop几乎没有任何安全机制，因此面临着各方面的安全威胁，主要包括以下几个方面。

（1）缺乏用户与服务之间的认证机制

1) NameNode或者JobTracker缺少用户认证机制：由于用户可以在应用程序中设置自己的用户名和所在的用户组，这使得任意用户可以很容易伪装成其他用户，给用户管理造成极大不便。

2) DataNode上缺少用户授权机制：DataNode上的Block读写无任何访问控制机制，以至于用户只要知道Block ID，就能够获取对应Block的内容，并且用户也可以随便往一个DataNode上直接写入Block。

3) JobTracker上缺少用户授权机制：

□任何一个用户均可以修改或者杀死其他用户的作业。

□任何一个用户均可以修改JobTracker的持久化状态。

（2）缺乏服务与服务之间的认证机制

DataNode与NameNode、TaskTracker与JobTracker之间缺少认证机制，以至于用户可以任意启动DataNode或者TaskTracker。

（3）缺乏传输以及存储加密措施

客户端与服务器之间、Slave与Master之间的数据传输采用TCP/IP协议，以Socket方式实现，但在传输和存储过程中没有加/解密处理。Hadoop各节点间的数据采用明文传输，使其极易在传输的过程中被窃取。

此外，数据服务器对内存和存储器中的数据没有任何存储保护，在恶意入侵、介质丢失、维修等情况下，数据容易泄露。

11.1.2 Hadoop对安全方面的需求

从2009年开始，Yahoo! 专门抽出一个团队解决Hadoop安全问题，对于前面介绍的各种问题，最终期望能够满足以下几个需求。

(1) 功能需求

- ❑ 引入授权机制，只有经授权的用户才可以访问Hadoop。
- ❑ 任何用户只能访问那些有权限访问的文件或者目录。
- ❑ 任何用户只能修改或者杀死自己的作业。
- ❑ 服务与服务之间引入权限认证机制，防止未经授权的服务接入Hadoop集群。
- ❑ 新引入的安全机制应对用户透明，且用户可放弃使用该机制以保证向后兼容。

(2) 性能需求

引入安全机制后带来的开销应在可接受范围内。

11.1.3 Hadoop安全设计基本原则

（1）作为一种可选方案

考虑到引入Hadoop安全机制会给运维人员带来一定的麻烦，且并不是所有Hadoop集群都需要安全机制，因此，Hadoop中的安全机制应是可配置的，默认情况下与现有方案（简单基于操作系统的认证机制）保持一致。

（2）无须显式输入密码

为了保证Hadoop的易用性，Hadoop的安全机制不应让用户显式地输入密码。

（3）保持向后兼容

当前很多应用程序运行于不同版本的Hadoop集群中，引入Hadoop安全机制后应不影响它们的功能，比如HFTP^[1]应可以在带和不带安全机制的集群中复制数据。

为了增强Hadoop的安全性，从2009年起，Apache专门抽出一个团队，为Hadoop增加安全认证和授权机制，在Apache Hadoop 1.0版本中加入了Kerberos^[2]身份认证和基于ACL（Access Control List）的服务访问控制机制。

[1] <http://hadoop.apache.org/docs/hdfs/current/hftp.html>

[2] <http://web.mit.edu/kerberos/>

11.2 基础知识

Hadoop RPC中采用了SASL（Simple Authentication and Security Layer，简单认证和安全层）进行安全认证，具体认证方法涉及DIGEST-MD5和Kerberos两种。本节详细介绍SASL、DIGEST-MD5和Kerberos的基本概念和原理。

11.2.1 安全认证机制

1.SASL

SASL是一种用来扩充C/S模式验证能力的认证机制。它的核心思想是把用户认证和安全传输从应用程序中隔离出来。比如SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）在定义之初没有考虑到用户认证等问题，现在SMTP可以使用SASL来完成这方面的工作。

SASL支持多种认证方法，主要包括以下几种。

□ANONYMOUS：无须认证。

□PLAIN：最简单的机制，但同时也是最危险的机制，因为信息采用明文密码方式传输。

□DIGEST-MD5：这是一种HTTP Digest兼容的安全机制，基于MD5，可以提供数据的安全传输层。这是方便性和安全性结合得最好的一种方式，也是SASL默认采用的方式。使用这种机制时，客户端与服务器端共享同一个密钥，而且该密钥不通过网络传输。验证过程是从服务器端先提出“质询”（实际上是一组信息）开始，客户端使用此“质询”与密钥计算出一个应答。不同的“质询”，不可能计算出相同的应答，且任何拥有密钥的一方，都可以用相同的“质询”算出相同的应答。因此，服务器端只要比较客户端返回的应答与自己算出的应答是否相同，就可以知道客户端所拥有的密钥是否正确。由于真正的密钥并没有通过网络进行传输，所以不怕网络窃取。

□GSSAPI：Generic Security Services Application Program Interface（通用安全服务应用程序接口）。GSSAPI本身是一套API，由IETF（Internet Engineering Task Force，互联网工程任务组）标准化。由于其最主要的实现是基于Kerberos的，所以一般说到GSSAPI都暗指Kerberos实现。我们将在下一小节对Kerberos进行详细介绍。

2.JAAS

JAAS（Java Authentication and Authorization Service，Java认证和授权服务）是SUN公司为了增强Java 2安全框架中的功能而提供的编程接口。Java 2安全框架提供的是基于代码源的存取控制方式，而JAAS还提供了基于代码运行者的存取控制能力，因此，JAAS是Java安全编程的一个重要补充。随着Internet安全问题越来越受到重视，JAAS在Java应用编程中得到了越来越广泛的应用。

JAAS主要由认证和授权两大部分构成。认证就是简单地对一个实体的身份进行判断；而授权则是向实体授予对数据资源和信息访问权限的决策过程。从1.4版本开始，JDK已经开始集成JAAS。

JAAS认证通过插件的形式工作，这使得Java应用程序独立于底层的认证技术，应用程序可以使用新的或经过修改的认证技术而不需要修改应用程序本身。应用程序通过实例化一个登录上下文对象来开始认证过程，这个对象根据配置决定采用哪个登录模块，而登录模块决定了认证技术和登录方式。一个比较典型的登录方式是提示输入用户名和口令，其他登录方式还有读入并核实声音或指纹样本。

JAAS的核心类可以分为公共类、认证类和授权类三部分。其中，公共类包括Subject、Principal、Credential三个类；认证类包

括LoginContext、LoginModule、CallbackHandler和Callback四个类；授权类包括Policy、AuthPermission和PrivateCrtx]entialPermission三个类。在Hadoop中仅用到了公共类和认证类，接下来我们分别对其进行介绍。

（1）公共类

公共类是由JAAS认证和授权的部分共同使用的类。其中Subject类最关键，它代表某个整体的一组相关信息，这个整体可以是一个人或其他对象，而相关信息则包括这个整体的标识（Principal）、公有凭据、私有凭据等。JAAS的标识类必须实现java.security.Principal接口，但凭据（Credential）类可以是任何对象。

Subject类

应用程序首先要对请求的来源进行认证，然后才能对该请求源访问的资源进行授权。JAAS框架定义“主题”这个术语代表请求源。主题可以是任何实体，如一个人或一项服务。一旦通过了认证，主题就和相关的标识联系在一起。标识可以使不同主题之间相互区别。一个主题可以具有多个标识，例如，一个人可以有名字标识和一个身份证号标识。主题也可以拥有与安全相关的属性，这些属性被称为凭据。需要特殊保护的凭据，如私钥，储存在私有凭据集合中；而可以公开的凭据，如公共证书，则储存在公共的凭据集合中。

主题类有以下几个比较重要的方法：

```
public Set getPrincipals(); //返回所有标识；
public static Object doAs (final Subject subject, final PrivilegedAction action);
//以某一主题身份执行action实例的run方法，如果正常执行，返回从run方法所返回的对象
```

Principal类

一个Principal对象可以和一个Subject对象关联，用于区别不同的主题（Subject）。用户自定义的Principal类必须实现Principal和Serializable接口。我们可以用Principal类提供的getPrincipals方法得到和更新与一个主题相联系的标识。

Credential类

Credential类通常用于实现一个凭据。凭据通常分为两种，分别是公共和私有凭据。核心JAAS类库没有对公共和私有凭据类做出规定，所以任何Java类都能代表凭据。但一般情况下，建议作为凭据的类应该实现Refreshable和Destroyable两个接口。Refreshable接口可以使凭据能够自我刷新，而Destroyable接口提供了销毁凭据中内容的功能。

（2）认证类

对一个主题进行认证需要以下步骤：

步骤1 应用程序实例化登录上下文（LoginContext）。

步骤2 登录上下文按照输入参数及配置装入所有相关的登录模块。

步骤3 应用程序调用登录上下文的Login方法。

步骤4 Login方法调用所有被装入的登录模块。每个模块都试图认证主题。如果认证成功，登录模块就把相关的标识和凭据关联到所认证的主题。

步骤5 登录上下文返回认证状态给应用程序。

步骤6 如果认证成功，应用程序可以从登录上下文中获得被认证的主题。

LoginContext类

LoginContext，即登录上下文类，给应用程序提供了认证主题的基本方法，并提供了一种独立于底层认证技术的应用程序开发方法。登录上下文依照配置决定应用程序采用哪些认证模块，这样，认证模块就以插件的形式工作于应用程序的底层，而认证技术的改变不需要修改应用程序本身。

LoginModule类

LoginModule类使开发者能够把不同类型的认证技术以插件的形式加到应用程序中。该类中最重要的方法是**login()**，定义如下：

```
boolean login() throws LoginException;
```

Login方法将由登录上下文自动调用，此时登录模块开始认证过程。

CallbackHandler类

登录过程的身份验证方式由用户所编写的**CallbackHandler**类决定。该类的实现既可以是提示用户输入用户名和密码，也可以是从智能卡或生物特征鉴别设备那里得到数据，或者简单地从底层的操作系统中抽取用户信息。

在某些情况下，登录模块必须和用户通信以获得认证信息，期间涉及的通信过程如下：

步骤1 应用程序提供一个实现**CallbackHandler**接口的类，并把该类实例的引用作为创建登录上下文对象时的参数。

步骤2 登录上下文接着把**CallbackHandler**类传给底层的登录模块。

步骤3 登录模块通过这个引用调用它的**Handle**方法，这样就可以从用户那里获取认证信息，也可以把信息送给用户。

由此可见，底层的登录模块与用户交互的方式完全是由应用程序决定的。

JAAS库中自带了一些认证机制，包括Windows NT、LDAP（**Lightweight Directory Access Protocol**，轻量目录访问协议）、Kerberos等，而Hadoop则主要采用了Kerberos。

11.2.2 Kerberos介绍

Kerberos是一种网络认证协议，主要用于计算机网络的身份鉴别。其特点是用户只需输入一次身份验证信息就可以凭借此验证获得的票据访问多个服务。Kerberos认证过程的实现不依赖于主机操作系统的认证。它不基于主机地址的信任，也不要求网络上所有主机的物理安全。Kerberos作为一种可信任的第三方认证服务，是通过传统的密码技术（如共享密钥）执行认证服务的。

（1）Kerberos协议中的基本概念

□客户端（Client）：客户端就是用户，也就是请求服务的用户。

□服务器（Server）：向用户提供服务的一方。

□密钥分发中心（Kerberos Key Distribution Center, KDC）：KDC存储了所有客户端密码和其他账户信息，它接收来自客户端的票据请求，验证其身份并对其授予服务票据。KDC中包含认证服务和票据授权服务两个服务。

□认证服务（Authentication Service, AS）：负责检验用户的身份。如果通过了验证，则向用户提供访问票据准许服务器的服务许可票据。

□票据授权服务（Ticket-Granting Service, TGS）：负责验证用户的服务许可票据。如果通过验证，则为用户提供访问服务器的服务许可票据。

□票据（Ticket）：用于在认证服务器和用户请求的服务之间安全地传递用户的身份，同时也传递一些附加信息。

□票据授权票据（Ticket-Granting Ticket, TGT）：客户访问TGS服务器需要提供的票据，目的是为了申请某一个应用服务器的服务许可票据。

□服务许可票据：客户请求服务时需要提供的票据。

（2）Kerberos认证过程介绍

Kerberos协议使用了对称加密机制，因此必然存在类似于口令的密钥。为了避免口令在网络中不必要的传递，Kerberos将服务认证功能交由认证服务和票据授权服务两个服务完成。为了阐述Kerberos中蕴含的设计思想，我们以多次出示信用卡消费密码购买电影票将提高泄露消费密码的几率为例进行类比。

用户Dong经常到电影院看电影。为了方便，他通常使用带有消费密码的信用卡购买电影票。为此，Dong需向电影院售票处出示自己的信用卡，并输入自己的信用卡密码才能购买所需的电影票，从而持票入场。检票员会验证Dong所持有的票的有效性，以决定是否允许Dong进电影院观看电影。如果Dong多次观看电影，则需多次出示自己的信用卡及密码购票。从概率学上讲，随着Dong观看电影次数的增多，其信用卡和密码被窃取的概率会增加。因此为了减少信用卡及密码被窃取的概率，Dong应该减少出示信用卡和密码的次数。

那么如何才能减少出示信用卡和密码的次数呢？解决方法之一就是在各个电影院之间引入专门销售“通用电影票”的售票机构。这样，Dong希望观看某场电影时，不再需要出示自己的信用卡和密码，而只需出示自己的通用景点票即可。比如“通用电影票”可能是Dong花费了500元购买的，可换取20场具体电影的一个凭证。因此，他的“通用电影票”并不是作为进入电影院的凭证，而是作为置换具体电影院对应电影票的凭证，即Dong只需向该电影院售票处出示该凭证，即可获取一张进入该电影院的电影票。使用“通用电影票”可避免多次出示信用卡及密码，从而减少了机密信息被窃取的可能性。在该解决方案中，存在两个核心机构：“通用电影票”售票机构和电影票售票处。

根据以上实例，我们可以引出Kerberos中的两个核心服务器：认证服务器和票据许可服务器，其中，认证服务器类似于“通用

电影票”售票机构，而票据许可服务器类似于电影票售票处。如同“通用电影票”售票机构不直接销售电影票，认证服务器也不直接颁发认证标识，而是只颁发可以购买认证标识的“票据”（TGT）。如同电影票售票处真正售票，票据许可服务器才真正地颁发认证标识。

在Kerberos中，当客户端请求一个服务时，Kerberos协议认证过程如图11-1所示，具体如下：

- 步骤1 客户端向KDC中的认证系统发送服务许可票据请求。
- 步骤2 KDC中的认证服务器接收到来自客户端的票据请求后，查找对应的数据，如果该用户合法，则为其返回服务许可票据。
- 步骤3 客户端向KDC中的票据许可服务器发送服务票据请求。
- 步骤4 票据许可服务器检查客户端的服务许可票据是否合法，如果合法，则为之返回服务票据。
- 步骤5 客户端获取服务许可票据后，向对应的服务器请求服务。
- 步骤6 服务器检查客户端的服务票据是否合法，如果合法，则为之返回服务器认证，从而可以安全地访问服务器。

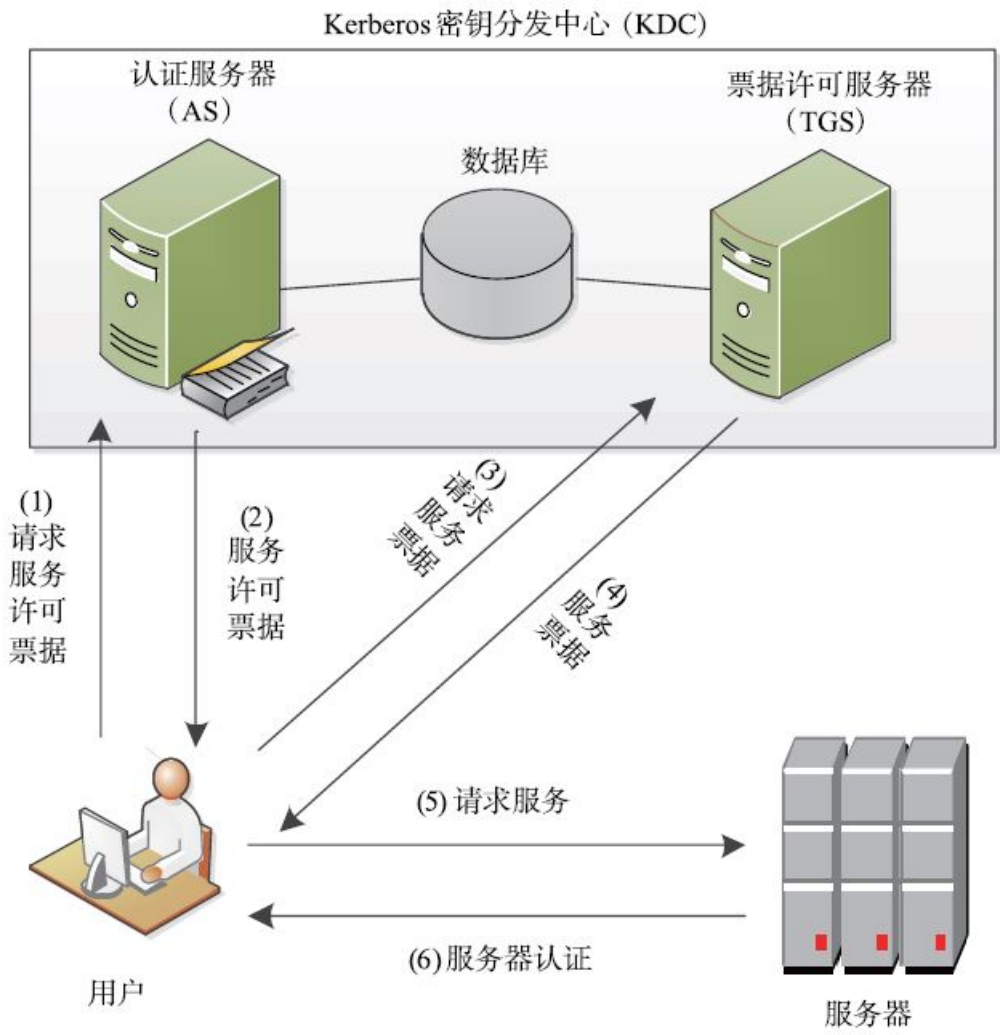


图 11-1 Kerberos认证过程

Hadoop选用了Kerberos作为安全认证机制。相比于另外一种常用机制SSL（Secure Sockets Layer），Kerberos具有以下两个优点。

□性能高：Kerberos采用了对称密钥，相比于SSL中自带的基于公钥的算法要高效得多。

□用户管理简单：Kerberos依赖于第三方的统一管理中心——KDC，管理员对用户的操作直接作用在KDC上，相比于SSL中基于广播的更新机制则简单得多。比如，撤销用户权限时只需将用户从KDC的数据库上删除即可，而在SSL中，需要重新生成一个证书撤销列表，并广播给各个服务器。

11.3 Hadoop安全机制实现

在1.0.0之后的版本中，Hadoop在RPC、HDFS和MapReduce等方面引入了安全机制。在本节中，我们将分别介绍RPC、HDFS和MapReduce中涉及的安全机制实现方法。

11.3.1 RPC

Hadoop RPC安全机制包括基于Kerberos和令牌的身份认证机制和基于ACL的服务访问控制机制。

1. 身份认证机制

为了保证网络通信的安全性，Hadoop中所有RPC连接均采用了SASL。在11.2.1小节中，我们已经介绍了SASL的原理。SASL本身不包含认证机制，需由用户指定一个第三方实现，而Hadoop正是将Kerberos和DIGEST-MD5两种认证机制添加到SASL中实现了RPC安全认证。在Hadoop中，除了NameNode，其他服务仅支持Kerberos认证方法。下面简要介绍这两种认证机制。

□ Kerberos：客户端（如JobClient）获取一个服务票据，然后才可以访问对应的服务器。

□ Kerberos+DIGEST-MD5：在这种机制中，Kerberos用于在客户端和服务端之间建立一条安全的网络连接，之后客户端可通过该连接从服务端获取一个密钥。由于该密钥仅有客户端和服务端知道，因此，接下来客户端可使用该共享密钥获取服务的认证。使用共享密钥进行安全认证（使用DIGEST-MD5协议）有多方面的好处：由于它只涉及认证双方而不必涉及第三方应用（比如Kerberos中的KDC），因此安全且高效；客户端也可以很方便地将该密钥授权给其他客户端，以让其他客户端安全访问该服务。我们将基于共享密钥生成的安全认证凭证称为令牌（Token）。在Hadoop中，所有令牌主要由identifier和password两部分组成，其中，identifier包含了该令牌中的基本信息，而password则是通过HMAC-SHA1作用在identifier和一个密钥上生成的，该密钥长度为20个字节并由Java的SecureRandom类生成。Hadoop中共有三种令牌，分别如下。

（1）授权令牌（Delegation Token）

授权令牌主要用于NameNode为客户端进行认证。当客户端初始访问NameNode时，如果通过Kerberos认证，则NameNode会为其返回一个密钥，之后客户端只需借助该密钥便可进行NameNode认证。为了防止重启后密钥丢失，NameNode将各个客户端对应的密钥持久化保存到镜像文件中。默认情况下，所有密钥每隔24小时更新一次，且NameNode总会保存前7小时的密钥以保证之前的密钥可用。

（2）数据块访问令牌（Block Access Token）

数据块访问令牌主要用于DataNode、SecondaryNameNode和Balancer为客户端存取数据块进行认证。当客户端向NameNode发送文件访问请求时，如果通过NameNode认证以及文件访问权限检查，则NameNode会将该文件对应的数据块位置信息和数据块访问密钥发送给客户端，客户端需凭借数据块访问密钥才可以读取一个DataNode上的数据块。NameNode会通过心跳将各个数据块访问密钥分发给DataNode、SecondaryNameNode和Balancer。需注意的是，数据块访问密钥并不会持久化保存到磁盘上，默认情况下，它们每隔10小时更新一次并通过心跳通知各个相关组件。

（3）作业令牌（Job Token）

作业令牌主要用于TaskTracker对任务进行认证。用户提交作业到JobTracker后，JobTracker会为该作业生成一个作业令牌，并写到该作业对应的HDFS系统目录下。当该作业的任务调度到各个TaskTracker上后，将从HDFS上获取作业令牌。该令牌可用于任务与TaskTracker之间进行相互认证（比如Shuffle阶段的安全认证）。与数据块访问令牌一样，作业令牌也不会持久化保存到内存中，一旦JobTracker重新启动，就会生成新的令牌。由于每个作业对应的令牌已经写入HDFS，所以之前的仍然可用。

相比于单纯使用Kerberos，基于令牌的安全认证机制有很多优势，具体如下。

❑性能：在Hadoop集群中，同一时刻可能有成千上万的任务正在运行。如果我们使用Kerberos进行服务认证，则所有任务均需要KDC中AS提供的TGT，这可能使得KDC成为一个性能瓶颈，而采用令牌机制则可避免该问题。

❑凭证更新：在Kerberos中，为了保证TGT或者服务票据的安全，通常为它们设置一个有效期，一旦它们到期，会对其进行更新。如果直接采用Kerberos验证，则需要将更新之后的TGT或者服务票据快速推送给各个Task，这必将带来实现上的烦琐。如果采用令牌，当令牌到期时，只需延长它的有效期而不必重新生成令牌。此外，Hadoop允许令牌在过期一段时间后仍可用，从而为过期令牌更新留下足够时间。

❑安全性：用户从Kerberos端获取TGT后，可凭借该TGT访问多个Hadoop服务，因此，泄露TGT造成的危害远比泄露令牌大。

❑灵活性：在Hadoop中，令牌与Kerberos之间没有任何依赖关系，Kerberos仅仅是进行用户身份验证的第一道防线，用户完全可以采用其他安全认证机制替换Kerberos。因此，基于令牌的安全机制具有更好的灵活性和扩展性。

2.服务访问控制机制

服务访问控制是Hadoop提供的最原始的授权机制，用于确保只有那些经过授权的客户端才能访问对应的服务。比如管理员可限制只允许若干用户/用户组向Hadoop提交作业。

服务访问控制是通过控制各个服务之间的通信协议实现的。它通常发生在其他访问控制机制之前，比如文件权限检查、队列权限检查等。

为了启用该功能，管理员需在core-site.xml中将参数hadoop.security.authorization置为true，并在hadoop-policy.xml中为各个通信协议指定具有访问权限的用户或者用户组。我们将具有访问权限的用户或者用户组称为访问控制列表（Access Control List, ACL）。管理员可为9个协议添加访问控制列表，如表11-1所示。

表 11-1 配置 Hadoop 的各个 ACL

属性名称	含 义
security.client.protocol.acl	ACL for ClientProtocol，用于控制访问 HDFS 的权限
security.client.datanode.protocol.acl	ACL for ClientDataNodeProtocol，客户端与 DataNode 之间的协议，主要用于 Block 恢复
security.datanode.protocol.acl	ACL for DataNodeProtocol，DataNode 与 NameNode 之间的通信协议
security.inter.datanode.protocol.acl	ACL for InterDataNodeProtocol，用于 DataNode 之间更新 Timestamp
security.namenode.protocol.acl	ACL for NameNodeProtocol，用于 Second NameNode 与 NameNode 之间通信
security.inter.tracker.protocol.acl	ACL for InterTrackerProtocol，用于 TaskTracker 与 JobTracker 之间通信
security.job.submission.protocol.acl	ACL for JobSubmissionProtocol，用于提交作业、查询作业状态等
security.task.umbilical.protocol.acl	ACL for TaskUmbilicalProtocol，用于 Task 与对应的 TaskTracker 通信
security.refresh.policy.protocol.acl	ACL for RefreshAuthorizationPolicyProtocol，用于更新 hadoop-policy.xml

这9个ACL的配置方法相同，即每个ACL可配置多个用户和用户组，用户之间和用户组之间都用“，”分割，而用户和用户组之间用空格分割。注意，如果只有用户组，前面必须保留一个空格，比如：

```
<property>
<name>security.job.submission.protocol.acl</name>
<value>alice, bob group1, group2</value>
</property>
```

上述代码表示用户alice和bob、用户组group1和group2可向Hadoop集群中提交作业。又如：

```
<property>
<name>security.client.protocol.acl</name>
<value>group3</value>
</property>
```

上述代码表示只有用户组group3可访问HDFS。

再如：

```
<property>
<name>security.client.protocol.acl</name>
<value>*</value>
</property>
```

上述代码表示所有用户和分组均可访问HDFS。

注意 默认情况下，这9个通信协议对任何用户和分组开放。

hadoop-policy.xml文件可使用以下命令动态加载。

❑更新NameNode相关配置：bin/hadoop dfsadmin-refreshServiceAcl。

❑更新JobTracker相关配置：bin/hadoop mradmin-refreshServiceAcl。

注意，只有属性security.refresh.policy.protocol.acl指定的用户才可以更新该配置文件。

客户端与HDFS之间的通信连接由两部分组成，它们均采用了Kerberos与令牌相结合的方法进行身份认证。

(1) 客户端向NameNode发起的RPC连接

由于NameNode存储了系统中所有文件的元数据信息，因此，如果客户端（如Task）需要读写一个文件，首先需要向NameNode发起RPC连接以获取文件所属的数据块列表。为了对客户端进行安全认证，Hadoop采用了Kerberos与授权令牌相结合的认证方法。

(2) 客户端向DataNode发起的Block传输连接

为了防止客户端随意从DataNode上读写数据，NameNode收到客户端发送的文件读写请求后，除了为之返回文件对应的Block列表外，还会为每个Block分配一个数据块访问令牌。这样，当客户端从DataNode读写Block时，首先需要出示待读写Block对应的访问令牌，只有通过DataNode验证后，才可以读写该Block。

下面重点介绍授权令牌和数据块访问令牌的身份认证过程。

(1) 授权令牌（Delegation Token）

当客户端初始访问NameNode时，需出示Kerberos票据获取NameNode认证。客户端通过认证后将收到一个授权令牌，之后便可以凭借该授权令牌访问NameNode。授权令牌可看作客户端与NameNode之间的共享密钥，当在不安全的链路上进行传输时应该保护好，任何人获取了该密钥都能够伪装成NameNode。另外，需要注意的是，只有通过Kerberos认证才可以获取授权令牌。

考虑到令牌的安全性，每个授权令牌均被赋予一定的有效期。当用户从NameNode上获取到一个授权令牌后，应告诉它谁是令牌的重新申请者。令牌的重新申请者应以自己的身份从NameNode端取得认证进而为用户更新令牌。更新令牌实际上就是延长令牌的NameNode上的有效期。在Hadoop中，所有令牌的重新申请者是JobTracker，它负责更新令牌直到作业运行完成。

认证过程

当一个客户端使用授权令牌向NameNode获取认证时，经过的步骤如下：

步骤1 客户端将TokenID发送给NameNode。其中TokenID定义如下：

```
TokenID={ownerID, renewerID, issueDate, maxDate, sequenceNumber}
```

步骤2 NameNodes使用TokenID和masterKey（NameNode和客户端共享masterKey），重新计算TokenAuthenticator和Token。其中TokenAuthenticator的计算方法如下：

```
TokenAuthenticator=HMAC-SHA1 (masterKey, TokenID)
```

步骤3 NameNode检查新的Token是否合法。一个Token是合法的，当且仅当Token在内存中存在，且当前时间仍在有效期内。其中Token计算方法如下：

```
DelegationToken={TokenID, TokenAuthenticator}
```

步骤4 如果Token是合法的，客户端和NameNode分别将TokenAuthenticator作为密钥、DIGEST-MD5作为认证协议进行双方认证。

重新申请令牌

考虑到令牌的安全性，Hadoop为每个授权令牌赋予一定的有效期，当令牌到期后，需指定一个令牌重新申请者。在Hadoop中，JobTracker是授权令牌的重新申请者。当JobTracker为客户端重新申请令牌时，它将旧令牌发送给NameNode，NameNode将检查TokenID中的信息，判断该令牌是否满足以下几个条件：

- JobTracker为该令牌的重新申请者；
- TokenAuthenticator正确；
- 当前时间currentTime小于最长有效期maxDate。

如果验证成功，则NameNode会延长该令牌的有效期。设延长时间为renewPeriod，则新的有效期为 $\max\{\text{currentTime} + \text{renewPeriod}, \text{maxDate}\}$ 。如果该Token不在内存中，说明NameNode可能因刚刚启动丢失了之前内存中所有的Token，则NameNode会将该Token重新添加到内存中。

NameNode会定期更新masterKey并保存到磁盘上，而Token则仅保存在内存中。

（2）数据块访问令牌（Data Block Access Token）

在原始Hadoop中，DataNode上没有任何针对Data Block的访问权限控制，这使得任何用户只要能够提供合法的Block ID便可以直接从DataNode上读取Data Block，用户也可以向DataNode上写入任意的Data Block。这主要是HDFS缺乏相关的安全机制造成的。当用户需要读取某个文件时，首先将请求发送给NameNode，NameNode检查该用户是否有该文件访问权限，如果有，则将该文件对应的Block元信息发送给用户，这样，用户再依次从对应的TaskTracker上读取所有Block便获取整个文件内容。由于DataNode上没有任何文件或者文件访问权限相关的概念，因此它不会对客户端的Block访问请求进行任何验证。

为了提供安全的数据读写机制，HDFS增加了基于块访问令牌的安全认证机制。块访问令牌由NameNode生成，并在DataNode端进行合法性验证。一个典型的应用场景如下：一个客户端向NameNode发送文件读请求，NameNode验证该用户具有文件读权限后，将文件对应的所有数据块的ID、位置以及数据块访问令牌发送给客户端；当客户端需要读取某个数据块时，将数据块ID和数据块访问令牌发送给对应的DataNode。由于NameNode已经通过心跳将密钥发送给各个DataNode，因此DataNode可以对数据块进行安全验证，而只有通过安全验证的访问请求才可以获取数据块。

客户端可以缓存来自NameNode的数据块访问令牌，且仅当令牌失效或令牌不存在时才从NameNode端重新获取。

11.3.3 MapReduce

MapReduce权限管理和身份认证涉及作业提交、作业控制、任务启动、任务运行和Shuffle等阶段。接下来分别对以上各阶段进行介绍。

(1) 作业提交

用户提交作业后，JobClient需与NameNode和JobTracker等服务进行通信，以进行身份认证和获取相关令牌，具体过程如下：

步骤1 JobClient与NameNode通信，通过Kerberos验证后可获取授权令牌。使用该令牌，作业和任务可以读写HDFS上的文件。

步骤2 JobClient将作业运行相关的文件，比如作业配置文件job.xml、输入分片相关文件job.split和job.splitmetainfo、程序jar包等，上传到HDFS的目录\${mapreduce.jobtracker.staging.root.dir}/\${user}/.staging/\${jobid}下（其中\${mapreduce.jobtracker.staging.root.dir}为\${hadoop.tmp.dir}/mapred/staging）。为了保证该目录中的文件不会被其他用户窃取，Hadoop将其访问权限设置为700。

步骤3 JobClient通过RPC将作业文件目录和授权令牌发送给JobTracker。

步骤4 JobTracker为作业生成作业令牌，连同HDFS的授权令牌，一并写入\${mapred.system.dir}/\${jobid}/job-info/jobToken中（\${mapred.system.dir}默认值为\${hadoop.tmp.dir}/mapred/system），同时将其访问权限设置为700。

注意 JobTracker对作业进行初始化时，需要从HDFS上读取job.splitmetainfo文件，此时，它是借用作业的授权令牌完成的。

(2) 作业控制

用户提交作业时，可通过参数mapreduce.job.acl-view-job指定哪些用户或者用户组可以查看作业状态，也可以通过参数mapreduce.job.acl-modify-job指定哪些用户或者用户组可以修改或者杀掉job。

(3) 任务启动

TaskTracker收到JobTracker分配的任务后，如果该任务来自某个作业的第一个任务，则会进行作业本地化：将任务运行相关的文件下载到本地目录下，其中，作业令牌文件会被写到\${mapred.local.dir}/tprivate/taskTracker/\${user}/jobcache/\${jobid}/jobToken目录下。由于只有该作业的拥有者可以访问该目录，因此令牌文件是安全的。此外，Task要使用作业令牌向TaskTracker进行安全认证，以请求新的任务或者汇报任务状态。

(4) 任务运行

增加安全机制之前，Hadoop中所有用户的任务均是以启动Hadoop的用户的身份启动的，也就是说，虽然各个用户以不同身份提交作业，但最终在各个节点上是以同一个用户运行身份运行的。很显然，这是不安全的，比如任何一个用户可以很容易杀掉另外一个用户的任务。

为了解决该问题，Hadoop应以实际提交作业的那个用户身份运行相应的任务。为此，Hadoop用C程序实现了一个setuid程序以修改每个任务所在JVM的有效用户ID。若要启用该功能，管理员首先需编译setuid程序，并将生成的可执行程序存放到\$HADOOP_HOME/bin目录下，然后在配置文件中将mapred.task.tracker.task-controller设置为org.apache.hadoop.mapred.LinuxTaskController，同时修改配置文件task-controller.cfg。

(5) Shuffle

在MapReduce中，一个作业的Map Task将结果直接写到TaskTracker的本地磁盘上，而Reduce Task则通过HTTP从TaskTracker上

获取数据。在添加安全机制之前，任何用户只要通过URL即可获得任意一个Map Task的中间输出结果，比如可使用以下URL获取节点node100上作业job_201211011150_10219的第0个Map Task的第0片数据：

```
http://node100: 33580/mapOutput?job=job_201211011150_10219&map=attempt_201211011150_10219_m_000000_0&
reduce=attempt_201211011150_10219_r_000000_0
```

为了解决该问题，Hadoop在Reduce Task与TaskTracker之间的通信机制上添加了双向认证机制，以保证有且仅有同作业的Reduce Task才能够读取Map Task的中间结果。该双向认证是以它们之间共享的作业令牌为基础的。

TaskTracker对Reduce Task认证

Reduce Task从TaskTracker上获取数据之前，先要将HMAC-SHA1（URL, JobToken）发送给TaskTracker, TaskTracker利用自己保存的作业令牌计算HMAC-SHA1，然后比较该值与Reduce Task发送过来的是否一致，如果一致，则通过身份认证。

Reduce Task对TaskTracker认证

为了防止伪装的TaskTracker向Reduce Task发送数据，Reduce Task也需要对TaskTracker进行认证。TaskTracker对Reduce Task认证成功后，需使用Reduce Task发送过来的HMAC-SHA1值与作业令牌计算一个新的HMAC-SHA1，经Reduce Task验证后，双方认证才算通过，此时才可以正式传送数据。

（6）Web UI

在Hadoop中，任何用户均可以通过Web界面观察整个集群的资源使用情况和每个作业的运行状态，这很显然缺乏必要的安全认证机制。考虑到Hadoop中的界面是基于Jetty实现的，因此，为了实现安全认证机制，需在Jetty中添加相应的模块。Kerberos中已经自带了Web浏览器访问认证机制SPNEGO（Simple and Protected GSS-API Negotiation）。Hadoop 1.0直接采用了该认证机制。

11.3.4 上层服务

在Hadoop中，很多上层服务充当Hadoop服务的请求代理，比如Oozie、Hive等。前面提到，Hadoop添加安全认证机制后，所有访问Hadoop的代理服务均需要拥有服务凭证。为此，Hadoop引入了“超级用户”的概念，这些用户可以以其他人的身份访问Hadoop的各个服务（类似于Linux中的sudo命令）。这些超级用户访问Hadoop服务时，首先自己需能够经过Kerberos认证，然后才能以其他用户的身份访问Hadoop服务。当超级用户希望与Hadoop的某个服务建立RPC连接时，首先需要使用doAs方法设置该连接；如果满足以下条件，Hadoop将接受连接请求：

□ 请求用户属于超级用户。

□ 用户所在的机器IP在安全IP列表内。

下面以Oozie为例说明上层服务安全访问Hadoop服务的方法。假设Hadoop中包含两个用户：超级用户oozie和普通用户dong。其中，oozie有Kerberos凭证，而dong没有。当用户将作业提交到oozie上后，oozie将以超级用户oozie身份进一步将作业提交到Hadoop上。而实际运行作业时，oozie想以普通用户dong的身份访问HDFS和MapReduce，也就是说，所有任务需以用户dong的身份启动，且以dong身份访问HDFS上文件，即用户oozie伪装成了用户dong。

（1）代码

为了让超级用户oozie安全地伪装成普通用户dong，需使用oozie的Kerberos凭证登录系统并为dong创建一个代理ugi，而真正的Hadoop服务访问操作则放在代理ugi的doAs方法中，具体如下：

```
.....
UserGroupInformation ugi=
UserGroupInformation.createProxyUser (user,
UserGroupInformation.getLoginUser());
ugi.doAs (new PrivilegedExceptionAction<Void> () {
public Void run()throws Exception{
//提交作业
JobClient jc=new JobClient (conf);
jc.submitJob (conf);
//访问HDFS
FileSystem fs=FileSystem.get (conf);
fs.mkdir (someFilePath);
}
}
```

（2）配置

管理员需要在配置文件中指定超级用户oozie可伪装成的所有用户，同时限制oozie所在的IP，具体如下：

```
<property>
<name>hadoop.proxyuser.oozie.groups</name>
<value>group1, group2</value>
<description>允许超级用户oozie冒充用户组group1和group2中的所有用户</description>
</property>
<property>
<name>hadoop.proxyuser.oozie.hosts</name>
<value>host1, host2</value>
<description>oozie只能从host1和host2上发起连接以冒充其他用户</description>
</property>
```

如果管理员不进行以上配置，则认为不允许任何用户伪装成其他用户。

11.4 应用场景总结

当管理员配置一个安全的Hadoop集群时，一般会创建两个用户：hdfs和mapreduce，并为其添加Kerberos认证，以让它们分别安全地启动HDFS服务和MapReduce服务。在一个安全的Hadoop集群中，各种应用场景涉及的安全认证过程如下。

11.4.1 文件存取

管理员需要在NameNode和DataNode所在节点上为用户hdfs添加Kerberos认证，这样，管理员以hdfs身份启动NameNode和DataNode，可避免非法NameNode或者DataNode接入Hadoop集群。

一个应用程序从HDFS上存取文件涉及的安全认证如图11-2所示。整个过程涉及Kerberos认证及Delegation Token和Block Access Token两种令牌，其中，Kerberos认证用于应用程序第一次访问NameNode时进行身份认证，通过认证之后，NameNode会为之分配Delegation Token或者Block Access Token，今后只需使用令牌访问NameNode或者从DataNode上存取数据即可。

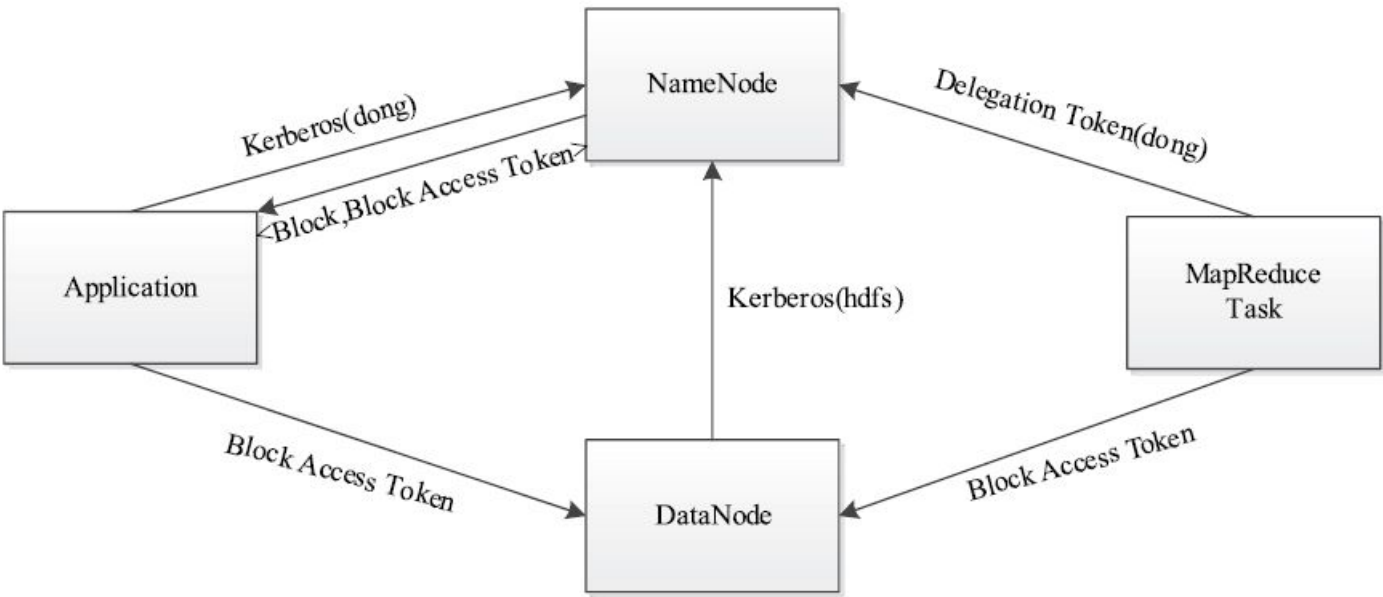


图 11-2 文件存取过程中涉及的安全认证

11.4.2 作业提交与运行

管理员需要在JobTracker和TaskTracker所在节点上为用户mapreduce添加Kerberos认证，这样，管理员以mapreduce身份启动JobTracker和TaskTracker，可避免非法JobTracker或者TaskTracker接入Hadoop集群。

一个应用程序从作业提交到运行涉及的安全认证如图11-3所示。整个过程涉及Kerberos认证及Delegation Token、Block Access Token、JobToken三种令牌。Kerberos认证、Delegation Token和Block Access Token的作用与11.4.1节中类似。JobToken创建于JobTracker端，并分发到各个TaskTracker以用于Task与TaskTracker之间的安全认证。

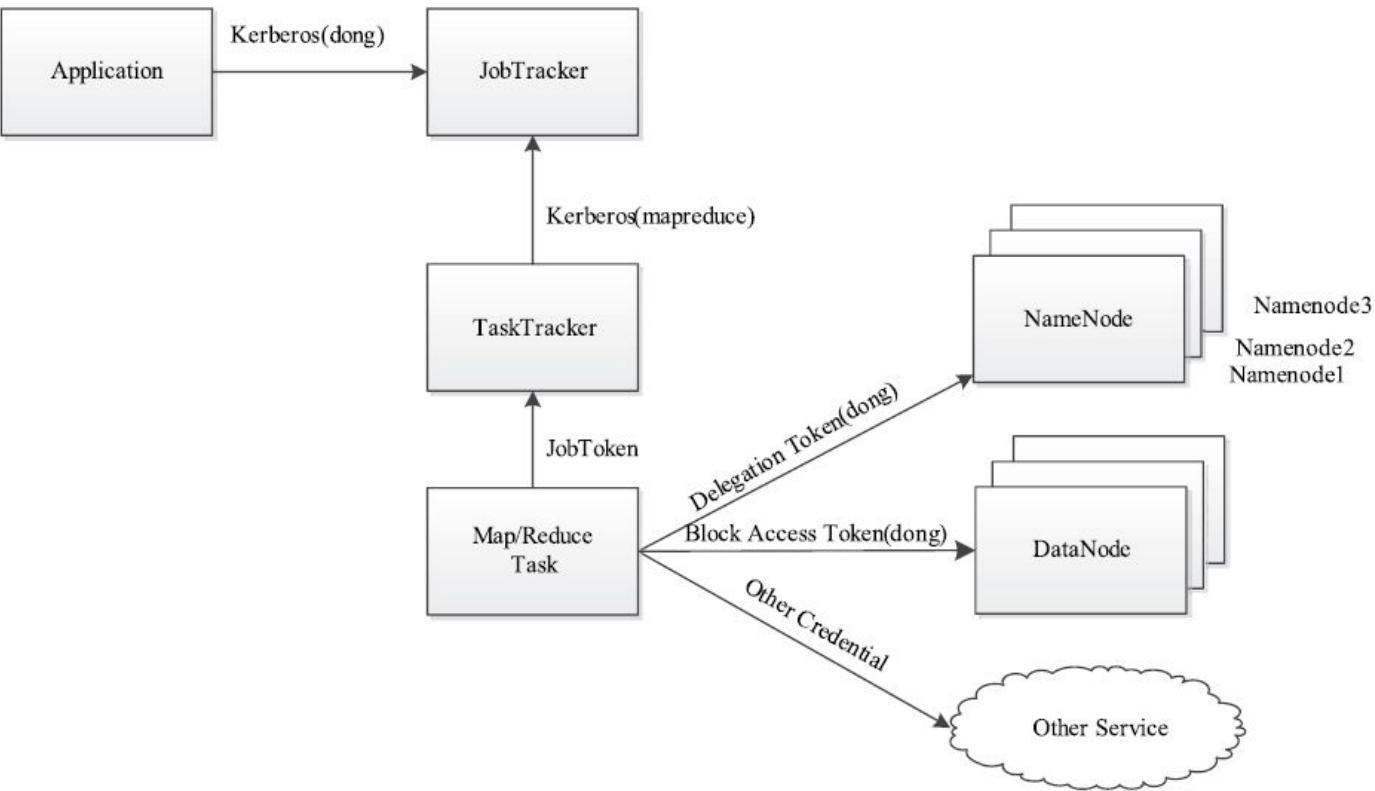


图 11-3 作业提交与运行过程中涉及的安全认证

11.4.3 上层中间件访问Hadoop

Hadoop有很多上层中间件，比如Oozie、Hive等。它们通常采用“伪装成其他用户”的方式访问Hadoop。以Oozie为例，其安全访问Hadoop流程如图11-4所示。超级用户oozie向Oozie提交作业，并要求伪装成普通用户dong在Hadoop上运行，而Oozie则进一步直接以超级用户oozie的身份将作业提交到Hadoop上，并要求以用户dong的身份运行作业但使用oozie的Kerberos凭证进行身份认证。

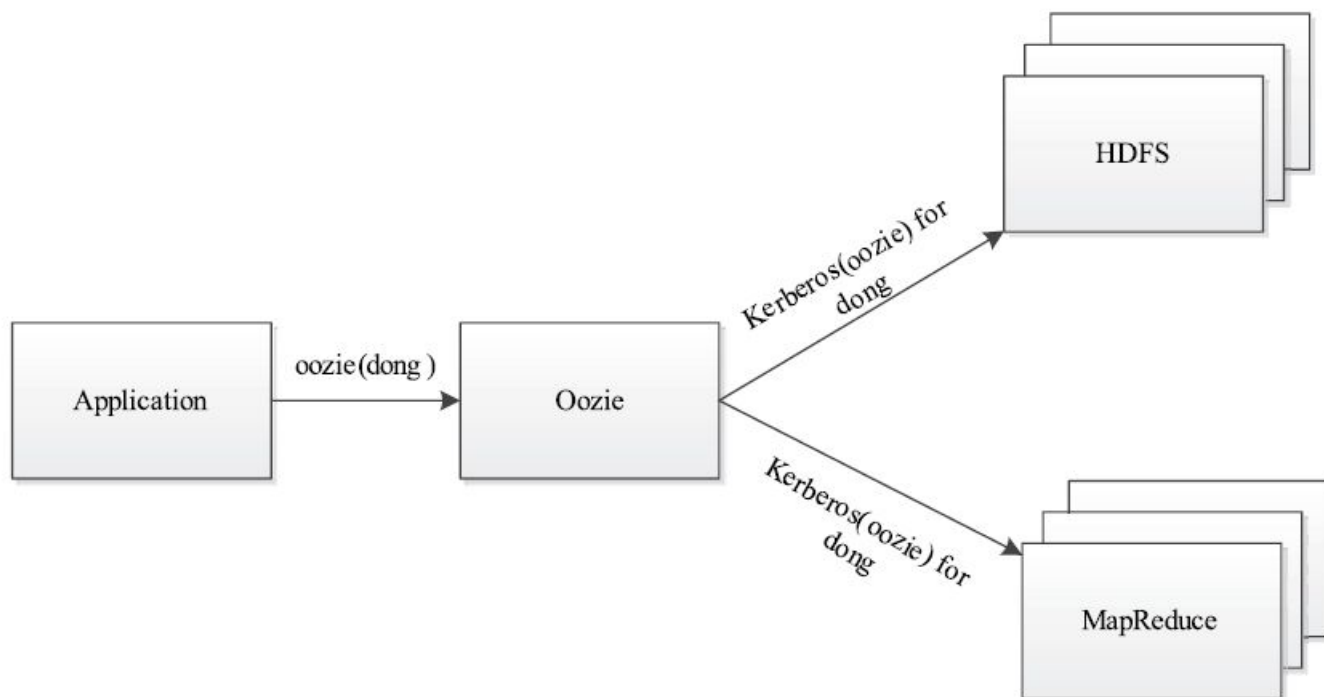


图 11-4 Oozie认证过程

11.5 小结

本章概括了旧版本（1.0之前版本）中Hadoop存在的安全问题，以及对安全的需求，并介绍了Hadoop 1.0中新加入的安全机制实现方案。

Hadoop 1.0加入了Kerberos与令牌相结合的身份认证和基于ACL（Access Control List）的服务访问控制机制。客户端第一次访问服务器端时需进行Kerberos身份认证，待通过认证后，服务器端会为之生成一个令牌，之后客户端只需凭借令牌便可以访问该服务器端；而ACL是Hadoop中采用的授权机制，管理员可为Hadoop中的任何服务配置允许访问的用户列表，以决定哪些用户可以访问对应的服务。

Hadoop安全机制是Hadoop 1.0中最重要的功能之一。它的引入标志着Hadoop已经真正成熟了。

第12章 下一代MapReduce框架

本书前面的章节主要介绍了第一代MapReduce框架（MapReduce Version 1.0，MRv 1）。随着时间的变迁，MRv 1已经变得日趋完善和稳定，且已被越来越多的公司采用。然而，随着数据量的高速增长和新型应用的出现，MRv 1在扩展性、可靠性、资源利用率和多框架支持等方面暴露出了明显不足，由此诞生了下一代MapReduce框架（MapReduce Version 2.0，MRv 2）。

本章将从基本设计思想、实现细节和 workflows 等方面对常见的MRv 2框架的开源实现进行介绍。

12.1 第一代MapReduce框架的局限性

MRv 1存在各种问题，主要可概括为以下几个方面。

□扩展性差：在MRv 1中，JobTracker同时具备了资源管理和作业控制两个功能，这成为系统的一个最大瓶颈，严重制约了Hadoop集群扩展性。

□可靠性差：MRv 1采用了master/slave结构，其中，master存在单点故障问题，一旦它出现故障，将导致整个集群不可用。

□资源利用率低：MRv 1采用了基于槽位的资源分配模型。槽位是一种粗粒度的资源划分单位，通常一个任务不会用完槽位对应的资源，且其他任务也无法使用这些空闲资源。此外，Hadoop将槽位分为Map slot和Reduce slot两种，且不允许它们之间共享，这常常会导致一种槽位资源紧张而另外一种闲置的情况出现（比如一个作业刚刚提交时，只会运行Map Task，此时Reduce slot闲置）。

□无法支持多种计算框架：随着互联网的高速发展，MapReduce这种基于磁盘的离线计算框架已经不能满足应用要求，从而出现了一些新的计算框架，包括内存计算框架、流式计算框架和迭代式计算框架等，而MRv 1不能支持多种计算框架并存。

为了克服以上几个缺点，Apache和Facebook均开始尝试对Hadoop进行升级改造，进而诞生了更加先进的下一代MapReduce计算框架。

12.2 下一代MapReduce框架概述

12.2.1 基本设计思想

在第6章中，我们已经介绍了JobTracker的基本功能，包括资源管理（由TaskScheduler模块实现）和作业控制（由JobTracker中多个模块共同实现）两部分，具体如图12-1所示。当前Hadoop MapReduce之所以在可扩展性、资源利用率和多框架支持等方面存在不足，正是由于Hadoop对JobTracker赋予的功能过多而造成负载过重。此外，从设计角度看，Hadoop未能够将资源管理相关的功能与应用程序相关的功能分开，造成Hadoop难以支持多种计算框架。

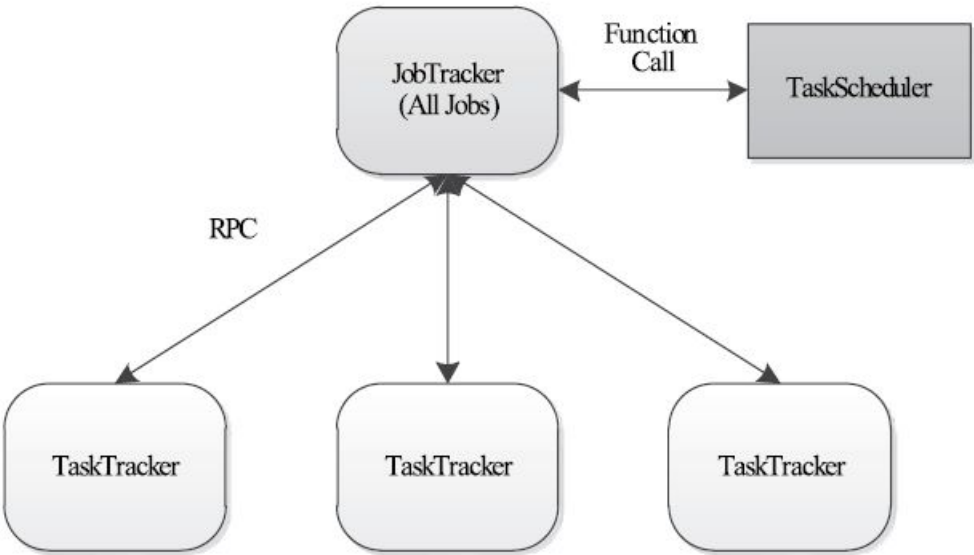


图 12-1 第一代MapReduce框架基本架构

下一代MapReduce框架的基本设计思想是将JobTracker的两个主要功能，即资源管理和作业控制（包括作业监控、容错等），分拆成两个独立的进程，如图12-2所示。资源管理进程是与具体应用程序无关的模块，它负责整个集群的资源（内存、CPU、磁盘等）管理；而作业控制进程则是直接与应用程序相关的模块，且每个作业控制进程只负责管理一个作业。这样，通过将原有JobTracker中与应用程序相关和无关的模块分开，不仅减轻了JobTracker负载，也使得Hadoop支持更多的计算框架。

12.2.2 资源统一管理平台

随着互联网的高速发展，基于数据密集型应用的计算框架不断出现。从支持离线处理的MapReduce，到支持在线处理的Storm，从迭代式计算框架Spark到流式处理框架S4.....各种框架诞生于不同的公司或者实验室。它们各有所长，各自解决了某一类应用问题。而在大部分互联网公司中，这几种框架可能同时被采用。比如在搜索引擎公司中，一种可能的技术方案如下：网页建索引采用MapReduce框架，自然语言处理/数据挖掘采用Spark（如网页PageRank计算、聚类分类算法等），对性能要求很高的数据挖掘算法用MPI等。考虑到资源利用率、运维成本、数据共享等因素，公司一般希望将所有这些框架部署到一个公共的集群中，让它们共享集群的资源，并对资源进行统一使用，这样，便诞生了资源统一管理与调度平台，如图12-3所示。资源统一管理与调度平台的典型代表是Apache的YARN（Yet Another Resource Negotiator）、Facebook的Corona和Berkeley的Mesos。

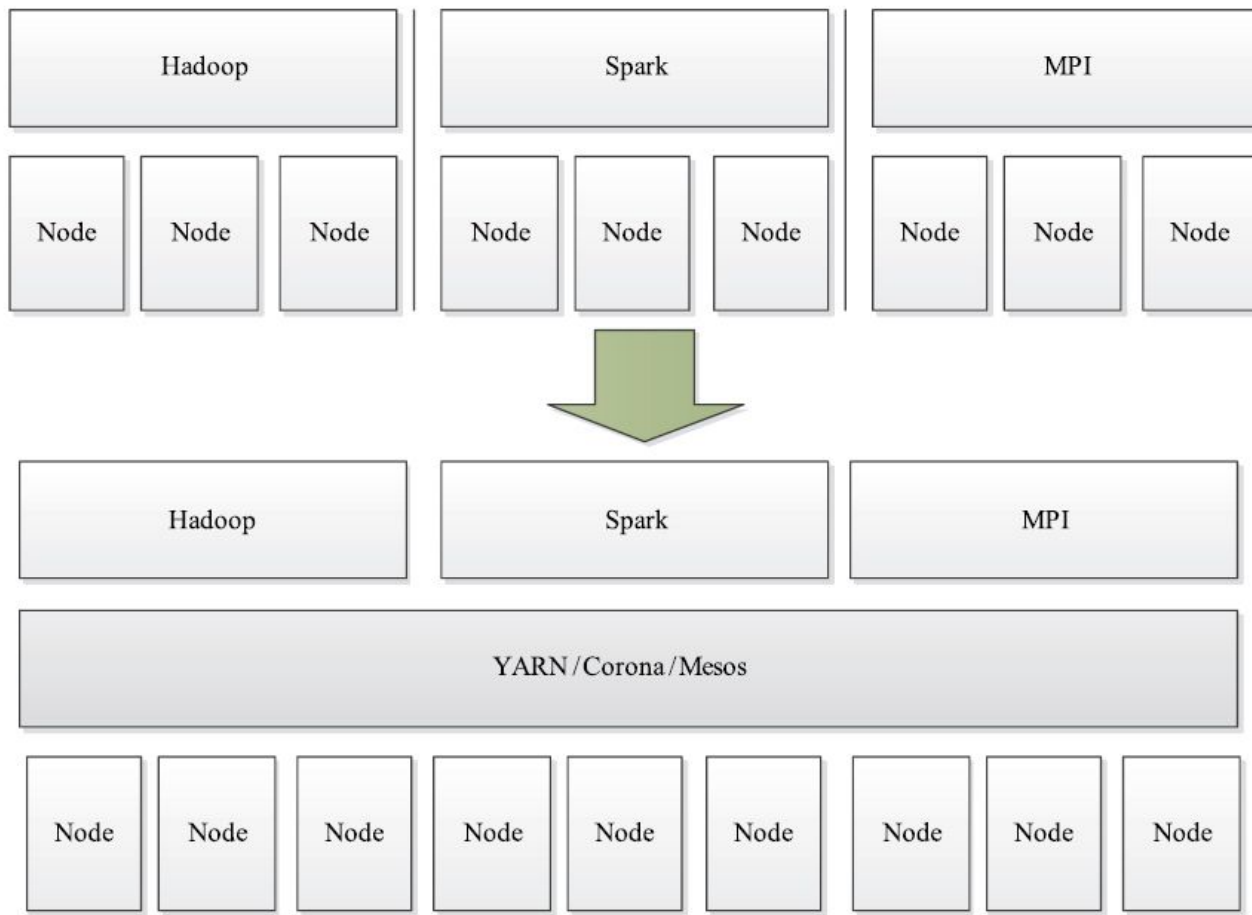


图 12-3 资源统一管理与调度平台的基本架构

从上面分析可知，MRv 2实际上是一个资源统一管理平台。它的目标已经不再局限于支持MapReduce一种计算框架，而是朝着对多种框架进行统一管理的发展方向。

相比于“一种计算框架一个集群”的模式，共享集群的模式存在多种好处。

❑资源利用率高：如图12-4所示，如果每个框架一个集群，则往往由于应用程序数量和资源需求的不均衡性，使得在某段时间内，有些计算框架的集群资源紧张，而另外一些集群资源空闲。共享集群模式则通过多种框架共享资源，使得集群中的资源得到更加充分的利用。

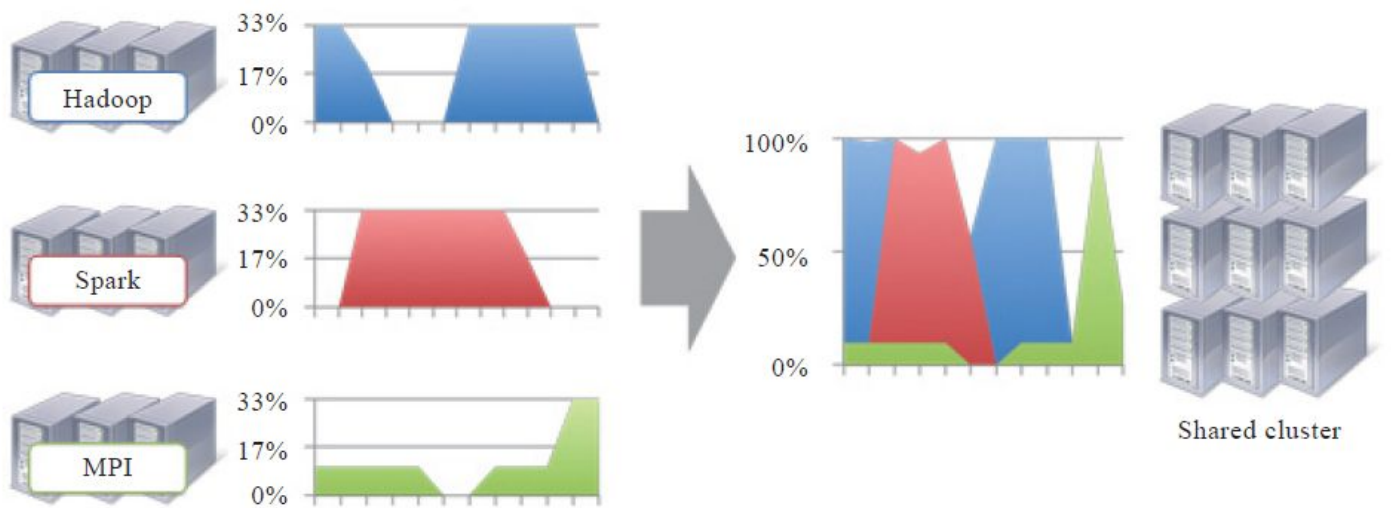


图 12-4 共享集群模式使得资源利用率提高

□ 运维成本低：如果采用“一个框架一个集群”的模式，则可能需要多个管理员管理这些集群，进而增加运维成本，而共享模式通常需要少数管理员即可完成多个框架的统一管理。

□ 数据共享：随着数据量的暴增，跨集群间的数据移动不仅需花费更长的时间，且硬件成本也会大大增加，而共享集群模式可让多种框架共享数据和硬件资源，将大大减少数据移动带来的成本。

12.3 Apache YARN

YARN是Apache的下一代MapReduce框架。它的基本设计思想是将JobTracker拆分成两个独立的服务：一个全局的资源管理器ResourceManager和每个应用程序特有的ApplicationMaster。其中，ResourceManager负责整个系统的资源管理和分配，而ApplicationMaster则负责单个应用程序的管理。

12.3.1 Apache YARN基本框架

YARN总体上仍然是master/slave结构。在整个资源管理框架中，ResourceManager为master, NodeManager为slave, ResourceManager负责对各个NodeManager上的资源进行统一管理和调度。当用户提交一个应用程序时，需要提供一个用于跟踪和管理这个程序的ApplicationMaster。它负责向ResourceManager申请资源，并要求NodeManager启动可以占用一定资源的任务。由于不同的ApplicationMaster分布在不同的节点上，因此它们之间不会相互影响。在本小节中，我们将从基本组成结构和RPC框架两方面对YARN进行介绍。

1.YARN的基本组成结构

图12-5描述了YARN的基本组成结构。YARN主要由ResourceManager、NodeManager、ApplicationMaster（图中给出了MapReduce和MPI两种计算框架的ApplicationMaster，分别为MR AppMstr和MPI AppMstr）和Container等几个组件构成。

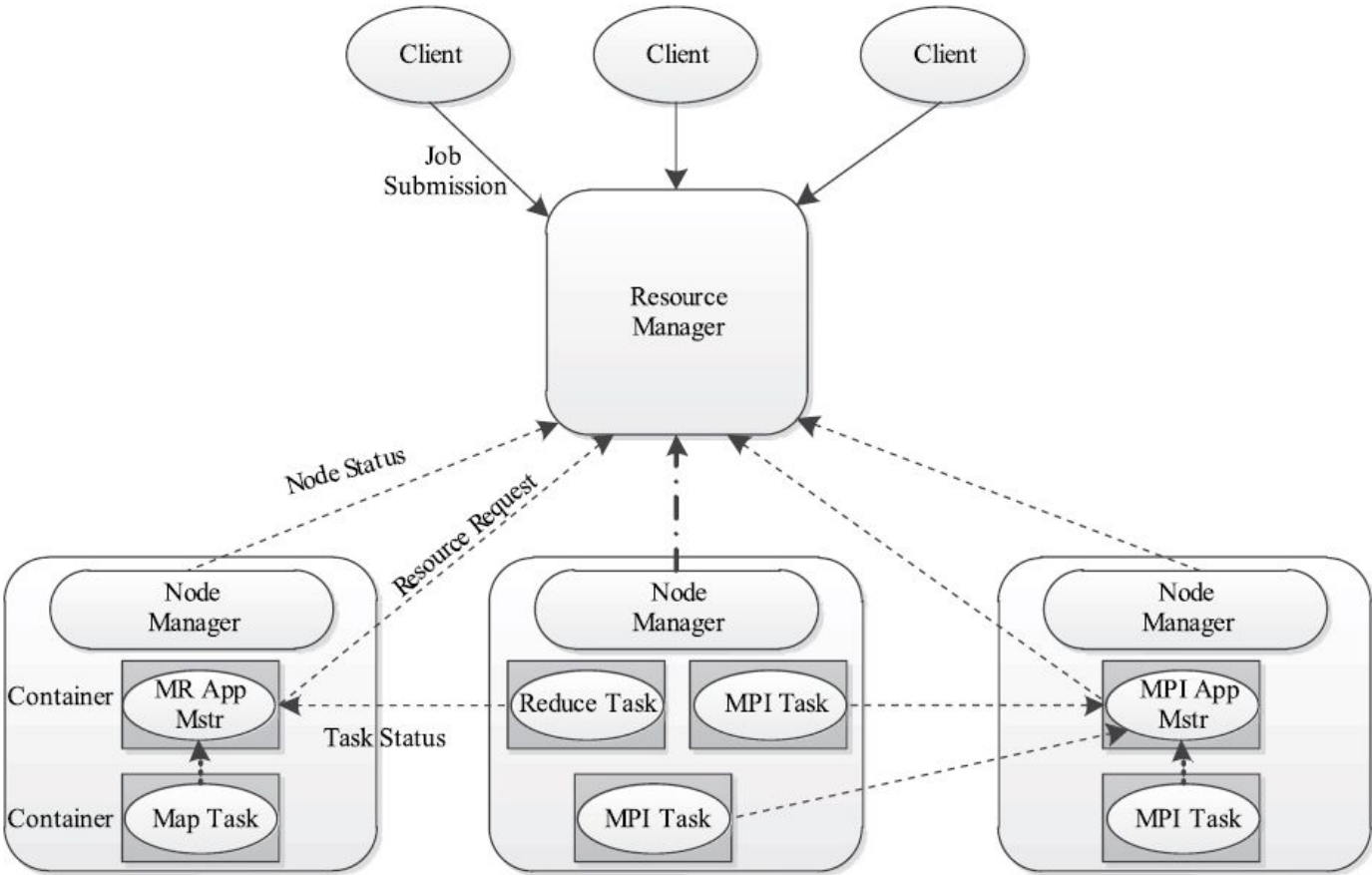


图 12-5 Apache YARN的基本架构

(1) ResourceManager (RM)

RM是一个全局的资源管理器，负责整个系统的资源管理和分配。它主要由两个组件构成：调度器（Scheduler）和应用管理器（Applications Manager, ASM）。

调度器

调度器根据容量、队列等限制条件（如每个队列分配一定的资源，最多执行一定数量的作业等），将系统中的资源分配给各个正在运

行的应用程序。需要注意的是，该调度器是一个“纯调度器”，它从事任何与具体应用程序相关的工作，比如不负责监控或者跟踪应用的执行状态等，也不负责重新启动因应用执行失败或者硬件故障而产生的失败任务，这些均交由应用程序相关的ApplicationMaster完成。调度器仅根据各个应用的资源需求进行资源分配，而资源分配单位用一个抽象概念“资源容器”（Resource Container，简称Container）表示。Container是一个动态资源分配单位，它将内存、CPU、磁盘、网络等资源封装在一起，从而限定每个任务使用的资源量。此外，该调度器是一个可插拔的组件，用户可根据自己的需要设计新的调度器，YARN提供了多种直接可用的调度器，比如Fair Scheduler和Capacity Scheduler等。

应用程序管理器

应用程序管理器负责管理整个系统中所有应用程序，包括应用程序提交、与调度器协商资源以启动ApplicationMaster、监控ApplicationMaster运行状态并在失败时重新启动它等。

为了保证高可用性，ResourceManager将所有状态信息保存到了Zookeeper^[1]中，它可以通过保存在Zookeeper中的状态快速重启。

（2）ApplicationMaster（AM）

用户提交的每个应用程序均包含一个AM。它实际上是一个简化版的JobTracker，主要功能包括：

- ❑ 与RM调度器协商以获取资源。
- ❑ 与NM通信以启动/停止任务。
- ❑ 监控所有任务的运行状态，并在任务运行失败时重新为任务申请资源以重启任务。

当前YARN自带了两个AM实现：一个是用于演示AM编写方法的实例程序distributedshell，它可以申请一定数目的Container运行一个shell命令或者shell脚本；另一个是运行MapReduce应用程序的AM——MRAppMaster，我们将在12.3.4节对其进行介绍。此外，一些其他的计算框架对应的AM正在开发中，比如Open MPI、Spark等^[2]。

（3）NodeManager（NM）

NM是每个节点上的资源和任务管理器。一方面，它会定时地向RM汇报本节点上的资源使用情况和各个Container的运行状态；另一方面，它接收并处理来自AM的任务启动/停止等各种请求。

（4）Container

Container是YARN中的资源分配单位，它封装了多维度的资源，如内存、CPU、磁盘、网络等。当AM向RM申请资源时，RM为AM返回的资源便是用Container表示的。YARN中每个任务均会对应一个Container，且该任务只能在该Container中执行，并仅能使用该容器代表的资源量。需要注意的是，Container不同于MRv1中的slot，它是一个动态资源划分单位，是根据应用程序的需求动态生成的。截至完成此书时，YARN仅支持CPU和内存两种资源，且使用了Linux Container进行资源隔离^[3]。

2.YARN的RPC协议与序列化框架

RPC协议是连接各个组件的“大动脉”，了解不同组件之间的RPC协议有助于我们更深入地学习YARN框架。在YARN中，任何两个需相互通信的组件之间仅有一个RPC协议，而对于任何一个RPC协议，通信双方有一端是Client，另一端为Server，且Client总是主动连接Server，因此，YARN实际上采用的是拉式（pull-based）通信模型。如图12-6所示，YARN主要由以下几个RPC协议组成。

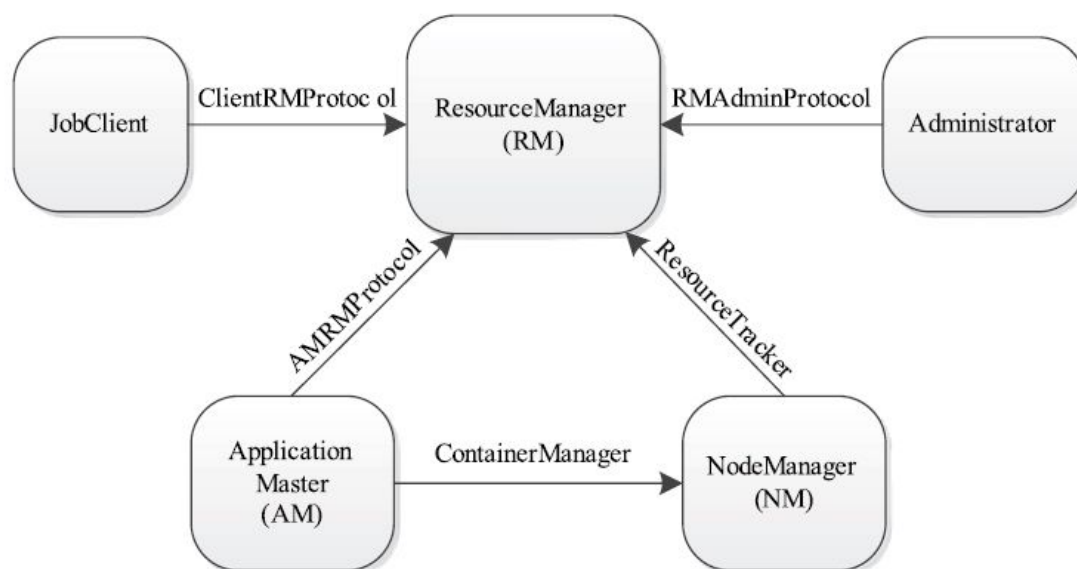


图 12-6 Apache YARN的RPC协议

□ Job Client（作业提交客户端）与RM之间的协议——ClientRMProtocol: Client通过该RPC协议提交应用程序，查询应用程序状态等。

□ Administrator（管理员）与RM之间的通信协议——RMAdminProtocol: Administrator通过该RPC协议更新系统配置文件，比如节点黑白名单、用户队列权限等。

□ AM与RM之间的协议——AMRMProtocol: Job AM通过该RPC协议向RM注册和撤销自己，并为各个任务申请资源。

□ AM与NM之间的协议——ContainerManager: AM通过该RPC协议要求NM启动或者停止Container，获取各个Container的使用状态等信息。

□ NM与RM之间的协议——ResourceTracker: NM通过该RPC协议向RM注册，并定时发送心跳信息汇报当前节点的资源使用情况和Container运行情况。

为了提高Hadoop的向后兼容性和不同版本之间的兼容性，YARN中的序列化框架采用了Google开源的Protocol Buffers。Protocol Buffers的引入使得YARN（与MRv 1相比）在兼容性方面向前迈进了一大步。

[1] <http://zookeeper.apache.org/>

[2] <http://wiki.apache.org/hadoop/PoweredByYarn>

[3] <https://issues.apache.org/jira/browse/YARN-3>

12.3.2 Apache YARN工作流程

当用户向YARN中提交一个应用程序后，YARN将分两个阶段运行该应用程序：第一个阶段是启动ApplicationMaster；第二个阶段是由ApplicationMaster创建应用程序，为它申请资源，并监控它的整个运行过程，直到运行成功。如图12-7所示，YARN的工作流程分为以下几个步骤：

步骤1 用户向YARN中提交应用程序，其中包括ApplicationMaster程序、启动ApplicationMaster的命令、用户程序等。

步骤2 ResourceManager为该应用程序分配第一个Container，并与对应的NodeManager通信，要求它在这个Container中启动应用程序的ApplicationMaster。

步骤3 ApplicationMaster首先向ResourceManager注册，这样，用户可以直接通过ResourceManager查看应用程序的运行状态；然后，它将为各个任务申请资源，并监控其运行状态，直到运行结束，即重复步骤4~7。

步骤4 ApplicationMaster采用轮询的方式，通过RPC协议向ResourceManager申请和领取资源。

步骤5 一旦ApplicationMaster申请到资源后，就与对应的NodeManager通信，要求其启动任务。

步骤6 NodeManager为任务设置好运行环境（包括环境变量、jar包、二进制程序等）后，将任务启动命令写到一个脚本中，并通过运行该脚本启动任务。

步骤7 各个任务通过某个RPC协议向ApplicationMaster汇报自己的状态和进度，以让ApplicationMaster随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务。

在应用程序运行的过程中，用户可随时通过RPC协议向ApplicationMaster查询应用程序的当前运行状态。

步骤8 应用程序运行完成后，ApplicationMaster向ResourceManager注销，并关闭自己。

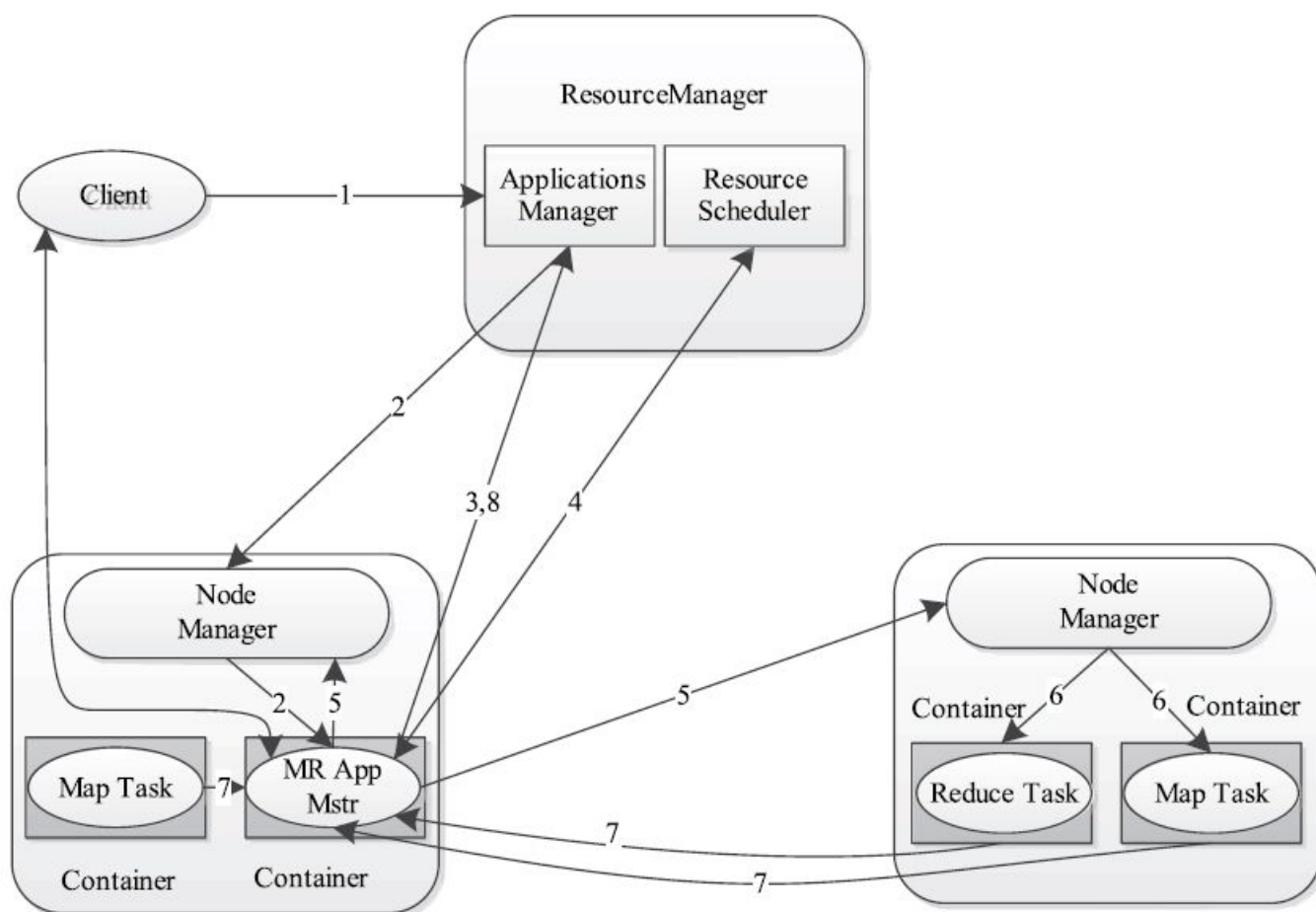


图 12-7 Apache YARN的工作流程

12.3.3 Apache YARN设计细节

Apache YARN在系统架构和软件设计等方面和MRv 1相比都有明显的改进。在前几节中，我们已经介绍了它的全新系统架构，而在本小节中，我们重点从软件设计和资源调度模型两方面介绍其采用的关键技术：在软件设计方面，YARN采用了基于服务的对象管理模型和基于事件驱动的并发模型；而在资源调度模型方面，YARN采用了更为细粒度的基于真实资源需求量的调度模型。

1.基于服务的对象管理模型

对于生命周期较长的对象，YARN采用了基于服务的对象管理模型对其进行管理。该模型主要有以下几个特点。

- ❑ 将每个被服务化的对象分为4个状态：NOTINITED（被创建）、INITED（已初始化）、STARTED（已启动）、STOPPED（已停止）。
- ❑ 任何服务状态变化都可以触发另外一些动作。
- ❑ 可通过组合的方式对任意服务进行组合，以便进行统一管理。

YARN中关于服务模型的类图如图12-8所示。在这个图中，我们可以看到，所有的服务对象最终均实现了接口Service，它定义了最基本的服务初始化、启动、停止等操作，而AbstractService类则提供了一个最基本的Service实现。YARN中所有对象，如果是非组合服务，则直接继承AbstractService类即可，否则需继承CompositeService。比如，对于ResourceManager而言，它是一个组合服务，它组合了各种服务对象，包括ClientRMService、ApplicationMasterLauncher、ApplicationMasterLauncher等。

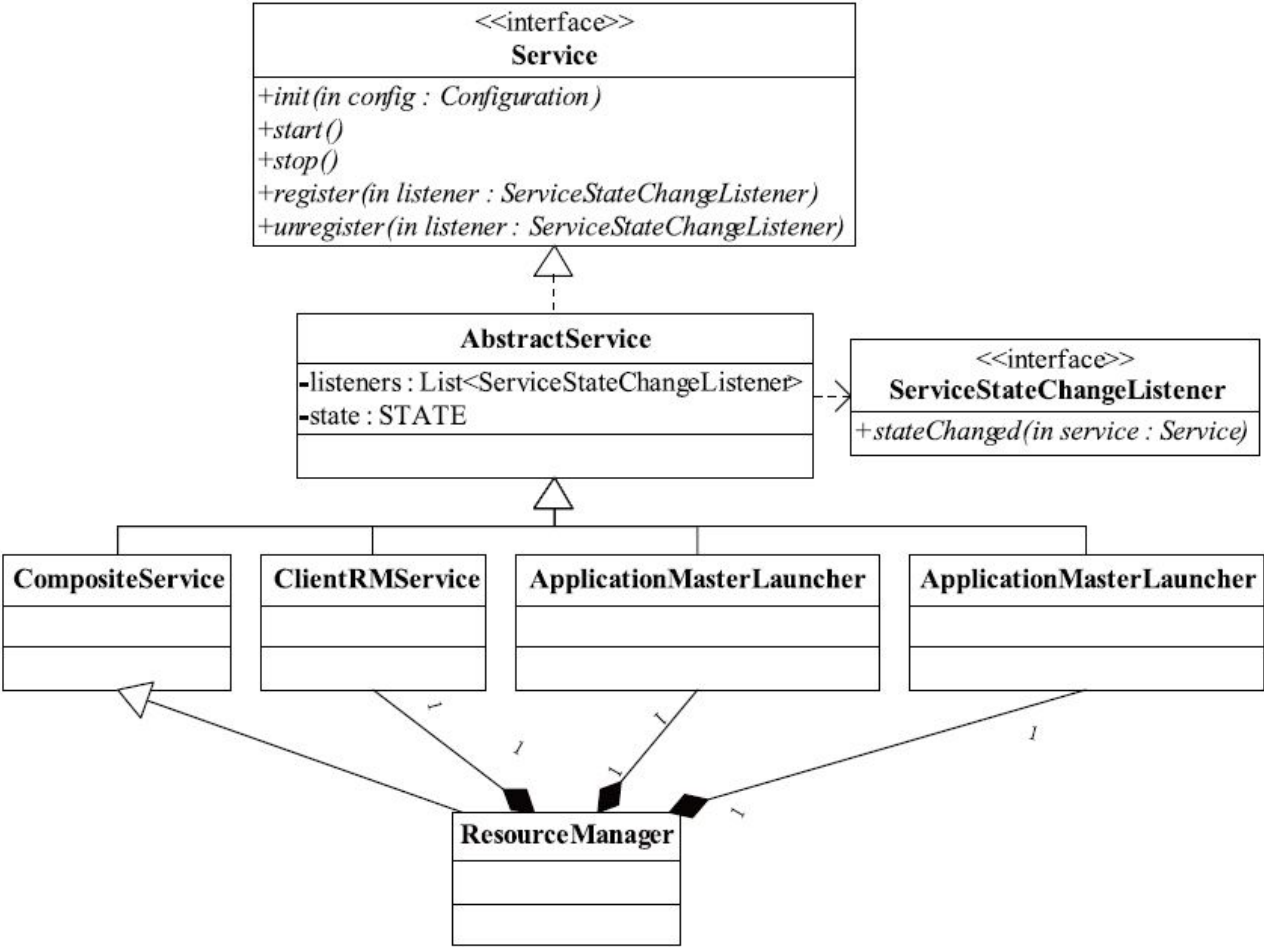


图 12-8 YARN中服务模型的类图

2.基于事件驱动的并发模型

YARN采用了基于事件驱动的并发模型。该模型能够大大增强并发性，从而提高系统整体性能。为了构建该模型，YARN将各种处理

逻辑抽象成事件和对应事件调度器，并将每类事件的处理过程分割成多个步骤，用有限状态机表示。YARN中的事件处理模型如图12-9所示。

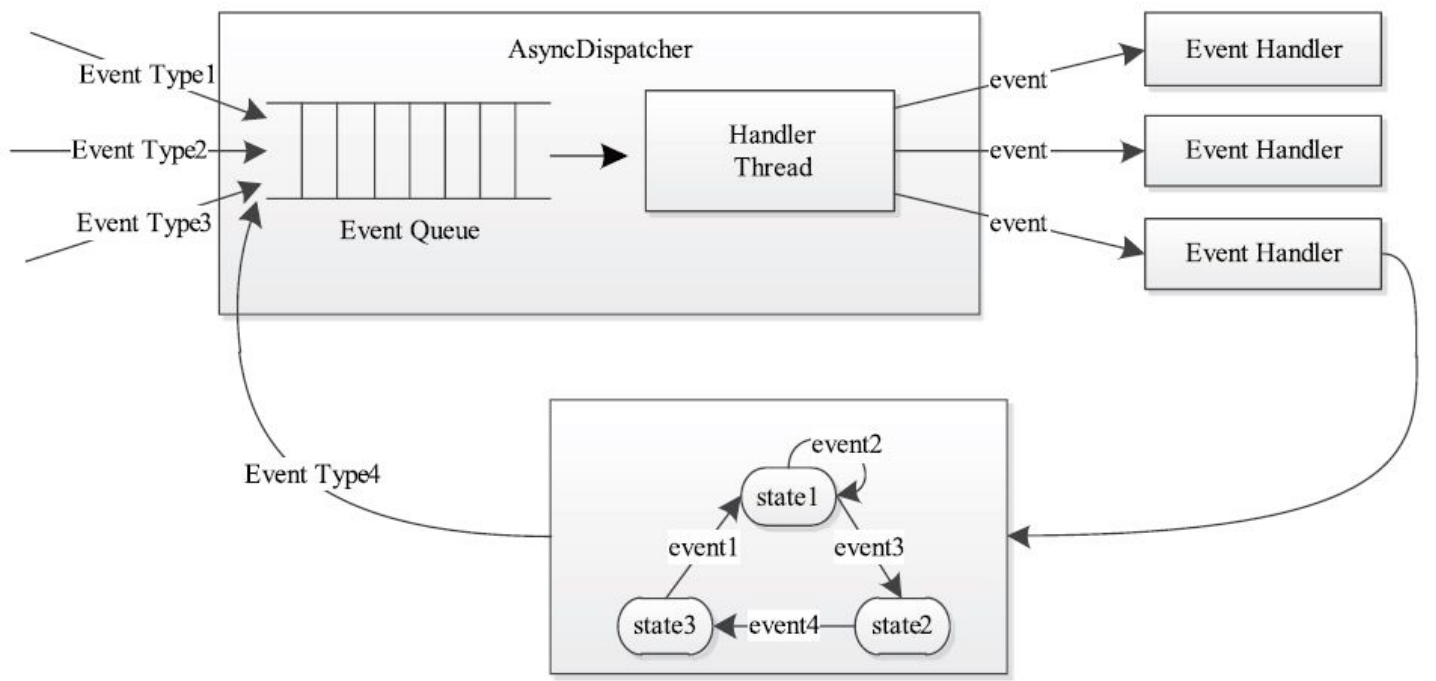


图 12-9 YARN的事件处理模型

整个处理过程大致为：处理请求会作为事件进入系统，由中央调度器（AsyncDispatcher）负责传递给相应事件调度器（Event Handler），该事件调度器可能将该事件转发给另外一个事件调度器，也可能交给一个带有有限状态机的事件处理器；其处理结果也以事件的形式输出给中央调度器，而新的事件会再次被中央调度器转发给下一个事件调度器，直至处理完成（达到终止条件）。

3.基于真实资源需求量的调度模型

不同于MRv 1中基于slot的资源模型，YARN采用了基于真实资源需求量的调度模型：集群中各个节点周期性向ResourceManager汇报各类资源使用情况（如CPU、内存等），而ResourceManager则根据各个作业的真实资源量需求，结合一些资源约束，进行资源分配和任务调度。

用户提交应用程序后，对应的ApplicationMaster负责将应用程序的资源需求转化成符合特定格式的资源请求，并发送给ResourceManager。一旦某个节点出现空闲资源，ResourceManager中的调度器将决定把这些空闲资源分配给哪个应用程序，并封装成Container返回给对应的ApplicationMaster。

ApplicationMaster发送的资源请求形式如下：

```
<Priority, Hostname, Resource, #Container>
```

上述的形式中涉及的几个属性的含义分别如下。

□Priority: 资源优先级。不同于MRv 1中的优先级概念（只有5种优先级，且难以扩展），YARN允许优先级是任意正整数，而具体怎样规划各种资源的紧急程度，完全由应用程序的ApplicationMaster决定。比如YARN自带的MapReduce框架的ApplicationMaster将资源分为三种优先级，分别是PRIORITY_FAST_FAIL_MAP（运行失败Map Task Attempt所需资源的优先级）、PRIORITY_REDUCE（Reduce Task所需资源的优先级）和PRIORITY_MAP（Map Task所需资源的优先级），优先级分别是5、10和20（Priority对应的数值越小，优先级越高）。ResourceManager将优先为优先级高的任务分配资源。

□Hostname: 期望分配的资源所在节点。调度器会优先为应用程序分配满足数据本地化的资源，这样可避免移动数据从而提高效率。而根据数据本地化的三个级别（Node-local、Rack-local和Off-switch），该属性有三个可选值：资源所在节点，资源所在机架和“*”。

其中，“*”表示可在任何节点上。

- Resource: 表示需要的资源量。截至2.0.3-alpha版本，YARN仅支持CPU和内存两种资源 [1]。
- #Container: 需要符合以上资源描述（优先级、所在节点和资源需求三方面的需求）的Container的数量。对于MapReduce而言，同一个作业的所有同类型任务需要的资源量是相同的。

【实例】“<10, nodeX, memory: 2GB|CPU: 1, 2>”表示请求2个满足以下条件的优先级为10的Container: 位于节点nodeX上、2 GB内存、1个CPU，这两个Container的优先级均为10。这意味着，如果没有优先级更高的Container需求，则需要优先分配这2个Container。

如果ApplicationMaster获取到了新资源，它会收到一个Container列表，其中每个Container主要包含以下内容：

<ContainerId, NodeId, NodeHttpAddress, Resource, ContainerToken>
--

当收到一个ID为ContainerId的Container后，ApplicationMaster将与ID为NodeId的NodeManager通信，并使用ContainerToken进行安全认证，以要求它启动一个占用Resource资源量的Container，在该Container中启动任务。与MRv 1只能够启动Java应用程序不同，YARN允许启动任意进程，比如Shell脚本、PHP进程、Python进程等。

[1] <https://issues.apache.org/jira/browse/YARN-2>

12.3.4 MapReduce与YARN结合

如果用户想要让一个新的计算框架运行在YARN上，需要将该框架重新封装成一个ApplicationMaster，而ApplicationMaster将作为用户应用程序的一部分被提交到YARN中。换句话说，YARN中的所有计算框架实际上只是客户端的一个库，因此，不必单独将这个框架以服务的形式部署到各个节点上。

由于YARN是直接从MRv 1衍化过来的，因此，它天生支持MapReduce计算框架。MapReduce框架对应的ApplicationMaster为MRAppMaster。接下来我们将详细对其进行介绍。

1.MRAppMaster基本构成

为了能够让MapReduce高效地运行在YARN之上，YARN采用异步事件模型对MRv 1中的几个重要数据结构进行了重写，使其更加高效。YARN对MRv 1的修改主要包括以下几方面：

□ 将JobTracker中的作业控制（如作业创建，监控作业运行状态等）部分拆分出来，按照规范实现MapReduce计算框架的ApplicationMaster—MRAppMaster。

□ TaskTracker的部分功能由模块TaskAttemptListenerImpl完成。

□ 利用状态机重写JobInProgress类，其主要功能由JobImpl完成。

□ 利用状态机重写TaskInProgress类，其主要功能由MapTaskImpl/ReduceTaskImpl完成。

新的MapReduce计算框架的所有功能都浓缩在了MRAppMaster中，其主要架构如图12-10所示。它主要由以下几个模块组成。

□ ContainerAllocator: ContainerAllocator负责将Map Task和Reduce Task需要的资源转化成ResourceManager可识别的表示形式，并通过AMRMPProtocol协议向Resource-Manager申请资源。

□ ContainerLauncher: MRAppMaster从ResourceManager端获取的资源是以Container表示的。Container中包含资源所在节点、资源种类以及每种资源的可用量等信息。ContainerLauncher通过ContainerManager协议与对应的节点通信，要求它启动任务。

□ TaskAttemptListener: 功能类似于MRv 1中的TaskTracker；不同的是，它不是管理某个节点上的任务，而是所有节点上的任务。它实现了TaskUmbilicalProtocol协议，所有任务周期性向其汇报运行状态。如果某个任务在一段时间内未发送心跳信息，它会将其杀死以重新计算。

□ JobTokenSecretManager: 管理作业令牌。作业令牌是Task与MRAppMaster通信，Reduce Task从Map Task拷贝数据的凭证，具体可参考第10章。

□ Task Cleaner: Task Cleaner负责清理失败或者被杀死任务产生的不完整输出结果，以防止大量无用的任务产生的结果塞满磁盘。

□ Speculator: 如果发现某个任务的执行速度落后于同作业的其他任务，则Speculator会为该任务启动一个备份任务，具体参考第6章。

□ Recovery Service: 当ApplicationMaster因失败重新启动后，Recovery Service可从日志中重构已经运行完成的任务的信息，进而避免重新计算这些任务。

□ MRClientService: MRClientService负责与客户端交互，可为客户端提供作业当前执行状态、进度等信息。

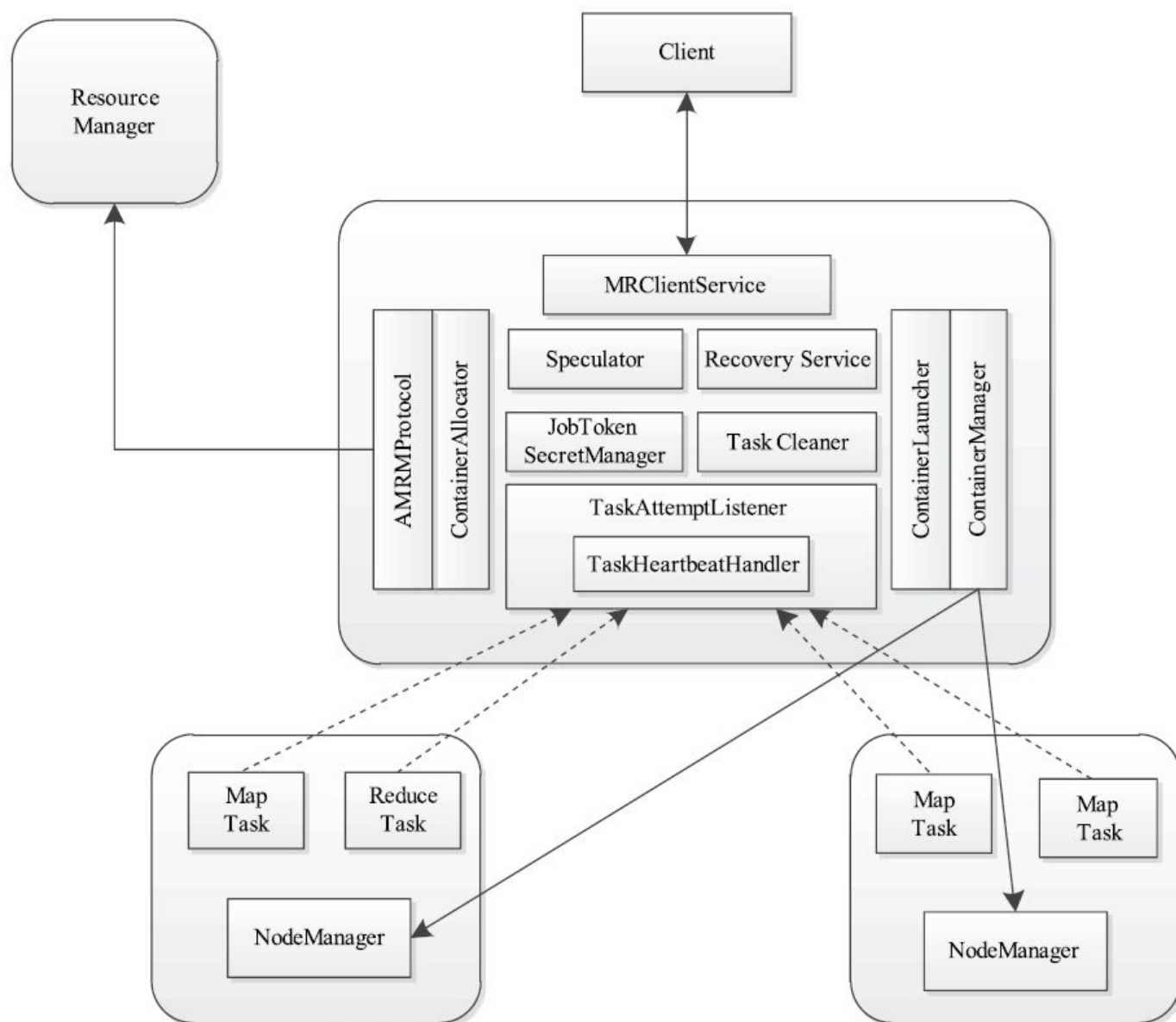


图 12-10 MRAppMaster的内部结构

2.MRAppMaster工作流程

按照作业大小不同，MRAppMaster提供了三种作业运行模式：本地模式（通常用于作业调试，与MRv1一样，不再赘述）、Uber模式^[1]和Non-Uber模式。对于小作业，为了降低其延迟，可采用Uber模式。在该模式下，所有Map Task和Reduce Task在同一个Container（MRAppMaster所在Container）中顺次执行。对于大作业，则采用Non-Uber模式。在该模式下，MRAppMaster先为Map Task申请资源，当Map Task运行完成的数目达到一定比例后再为Reduce Task申请资源。

（1）Uber运行模式

为了降低小作业延时，YARN专门对小作业运行方式进行了优化。对于小作业而言，MRAppMaster无须再为每个任务分别申请资源，而是让其重用同一个Container，并按照先Map Task后Reduce Task的运行方式串行执行每个任务。在YARN中，如果一个MapReduce作业同时满足以下条件，则认为是小作业，可运行在Uber模式下：

- Map Task数目不超过`mapreduce.job.ubertask.maxmaps`（默认是9）。
- Reduce Task数目不超过`mapreduce.job.ubertask.maxmaps`（默认是1）。
- 输入文件大小不超过`mapreduce.job.ubertask.maxbytes`（默认是一个Block大小）。
- Map Task和Reduce Task需要的资源量不超过MRAppMaster可使用的资源量。

另外，由于链式作业会并发执行Map Task和Reduce Task，因此不允许运行在Uber模式下。

（2）Non-Uber运行模式

在大数据环境下，Uber运行模式通常只能覆盖到一小部分作业，而对于其他大多数作业，仍将运行在Non-Uber模式下。在Non-Uber模式下，MRAppMaster将一个作业的Map Task和Reduce Task分为以下四种状态。

- pending: 刚启动但尚未向ResourceManager发送资源请求。
- scheduled: 已经向ResourceManager发送资源请求但尚未分配到资源。
- assigned: 已经分配到了资源且正在运行。
- completed: 已经运行完成。

对于Map Task而言，它的生命周期为scheduled→assigned→completed；而对于Reduce Task而言，它的生命周期为pending→scheduled→assigned→completed。由于Reduce Task的执行依赖于Map Task的输出结果，因此，为避免Reduce Task过早启动而造成资源利用率低下，MRAppMaster让刚启动的Reduce Task处于pending状态，以便能够根据Map Task运行情况决定是否对其进行调度。在YARN之上运行MapReduce作业需要解决两个关键问题：如何确定Reduce Task启动时机以及如何完成Shuffle功能。

如何确定Reduce Task启动时机

由于YARN中不再有Map slot和Reduce slot的概念，且RedouceManager也不知道Map Task与Reduce Task之间存在依赖关系，因此，MRAppMaster自己需设计资源申请策略以防止因Reduce Task过早启动而造成资源利用率低下和Map Task因分配不到资源而“饿死”。MRAppMaster在MRv1原有策略（Map Task完成数目达到一定比例后才允许启动Reduce Task）基础上添加了更为严格的资源控制策略和抢占策略。总结起来，Reduce Task启动时机由以下三个参数控制。

- `mapreduce.job.reduce.slowstart.completedmaps`: 当Map Task完成的比例达到该值后才会为Reduce Task申请资源，默认是0.05。
- `yarn.app.mapreduce.am.job.reduce.rampup.limit`: 在Map Task完成前，最多启动的ReduceTask比例，默认为0.5。
- `yarn.app.mapreduce.am.job.reduce.preemption.limit`: 当Map Task需要资源但暂时无法获取资源（比如Reduce Task运行过程中，部分Map Task因结果丢失需重算）时，为了保证至少一个Map Task可以得到资源，最多可以抢占的Reduce Task比例，默认为0.5。

如何完成Shuffle功能

按照MapReduce的基本逻辑，Shuffle HTTP Server应该分布到各个节点上，以便能够支持各个Reduce Task远程拷贝数据。然而，由于Shuffle是MapReduce框架中特有的一个处理流程，从设计上讲，不应该将它直接嵌到YARN的某个组件（比如NodeManager）中。

前面提到，YARN采用了服务模型管理各个对象，且多个服务可以通过组合的方式交由一个服务进行统一管理。在YARN中，NodeManager作为一种组合服务模式，允许动态加载应用程序临时需要的附加服务。利用这一特性，YARN将Shuffle HTTP Server组装成了一种服务，以便让各个NodeManager加载它。

[1] <https://issues.apache.org/jira/browse/MAPREDUCE-2405>

12.4 Facebook Corona

Corona^[1]是Facebook于2012年11月开源的下一代MapReduce框架。它的设计目标与YARN类似，在Facebook团队的博文^[2]中如此描述Corona的设计目标：

- 更好的可扩展性和集群资源利用率。
- 更小的短作业运行延迟。
- 支持在线版本更新。
- 基于真实资源需求进行任务调度。

与YARN最大的不同是，Corona暂时未考虑对多计算框架的支持，也就是说，Corona的目标限定在MapReduce一种计算框架中（尽管在源代码中将Corona和MapReduce代码分开存放到不同的jar包中，但它们还是耦合在一起的）。

接下来将重点介绍Corona的基本框架和工作原理。

12.4.1 Facebook Corona基本框架

与YARN一样，Corona也采用了master/slave结构。它们的基本思想也是一致的，即将JobTracker拆分成了两个独立的服务：全局的资源管理器ClusterManager和每个应用程序特有的CoronaJobTracker。其中，ClusterManager负责整个系统的资源管理和分配，而CoronaJobTracker则负责单个应用程序的管理。

1. Corona的基本组成结构

（1）ClusterManager（CM）

类似于YARN中的ResourceManager，负责系统资源分配和调度。ClusterManager掌握着各个节点的资源使用情况，并将资源分配给各个应用程序（采用的调度器为Fair Scheduler）。此外，考虑到在新架构中，ClusterManager本身出故障的可能性非常低，它的高可用性通常仅用于在线升级，因此，Corona在ClusterManager高可用性方面设计的非常简单：仅支持人工触发命令将ClusterManager状态信息保存到一个镜像文件中，并在升级结束后将其从该文件中恢复。

（2）CoronaJobTracker（CJT）

类似于YARN中的ApplicationMaster，用于MapReduce应用程序的监控和容错。在Corona中，CoronaJobTracker可以运行在三个模式下。

- 同进程模式：运行在客户端中，该模式能大大降低小作业运行延迟。
- 转发模式：将来自客户端的请求转发给位于远程模式下的CoronaJobTracker。
- 远程模式：运行在某个CoronaTaskTracker上，它最重要的任务是为作业申请资源和监控作业的运行过程，直到它运行结束。

与MRv 1中的JobTracker不同，每个CoronaJobTracker只负责管理一个作业。

（3）CoronaTaskTracker（CTT）

功能类似于YARN中的NodeManager，它的实现重用了MRv 1中TaskTracker的很多代码（实际上，CoronaTaskTracker类继承了MRv 1的TaskTracker类）：一方面，它通过心跳将节点资源使用情况汇报给ClusterManager；另一方面，它与CoronaJobTracker通信，以获取待运行的任务和汇报正运行任务的状态。

(4) ProxyJobTracker

ProxyJobTracker是一种离线查看作业运行信息的工具。当一个作业正在运行时，用户可通过CoronaJobTracker提供的Web服务查看作业的当前运行情况；而一旦作业运行完成或者节点处于离线状态时，用户则可通过ProxyJobTracker查看作业的详细运行信息，比如作业的各个任务执行情况、作业Metrics、作业Counter等。

2. Corona的RPC协议与序列化框架

Hadoop Corona是基于Hadoop 0.20版本实现的，所以Hadoop 0.20中的RPC协议在Corona中仍被采用，包括客户端与JobTracker通信协议JobSubmissionProtocol、TaskTracker与JobTracker通信协议InterTrackerProtocol和Task与TaskTracker之间的通信协议TaskUmbilicalProtocol^[3]。这几个协议在第4章均有介绍，在此不再赘述。除了在MRv1中已存在的这些协议，Corona又增加了一些其他协议，如图12-11所示。

□ ClusterManagerService: 它是采用Thrift RPC实现的协议，主要用于CoronaJobTracker、CoronaTaskTracker与ClusterManager通信。它定义了CoronaJobTracker申请和释放资源、CoronaTaskTracker汇报心跳等RPC接口。

□ SessionDriverService: 该协议也是采用Thrift RPC实现的。ClusterManager通过该协议主动将信息（如新分配的资源、待释放的资源等）推送给CoronaJobTracker（而不是让CoronaJobTracker轮询获取这些信息）以降低延迟。

□ CoronaTaskTrackerProtocol: 该协议是采用Hadoop自带的RPC框架实现的。CoronaJobTracker可通过该协议主动将命令（如启动新任务、杀死任务等命令）推送给CoronaTaskTracker以降低延迟（而不是像MRv1那样，TaskTracker通过心跳周期性拉取JobTracker命令）。

□ InterCoronaJobTrackerProtocol: 该协议是采用Hadoop自带的RPC框架实现的，远程模式下的CoronaJobTracker刚启动时，将通过该协议向客户端（父CoronaJobTracker）汇报它所在节点的host和端口号，此后它还会周期性向客户端汇报心跳，以探测父CoronaJobTracker是否活着，一旦发现父CoronaJobTracker失败，则直接退出。

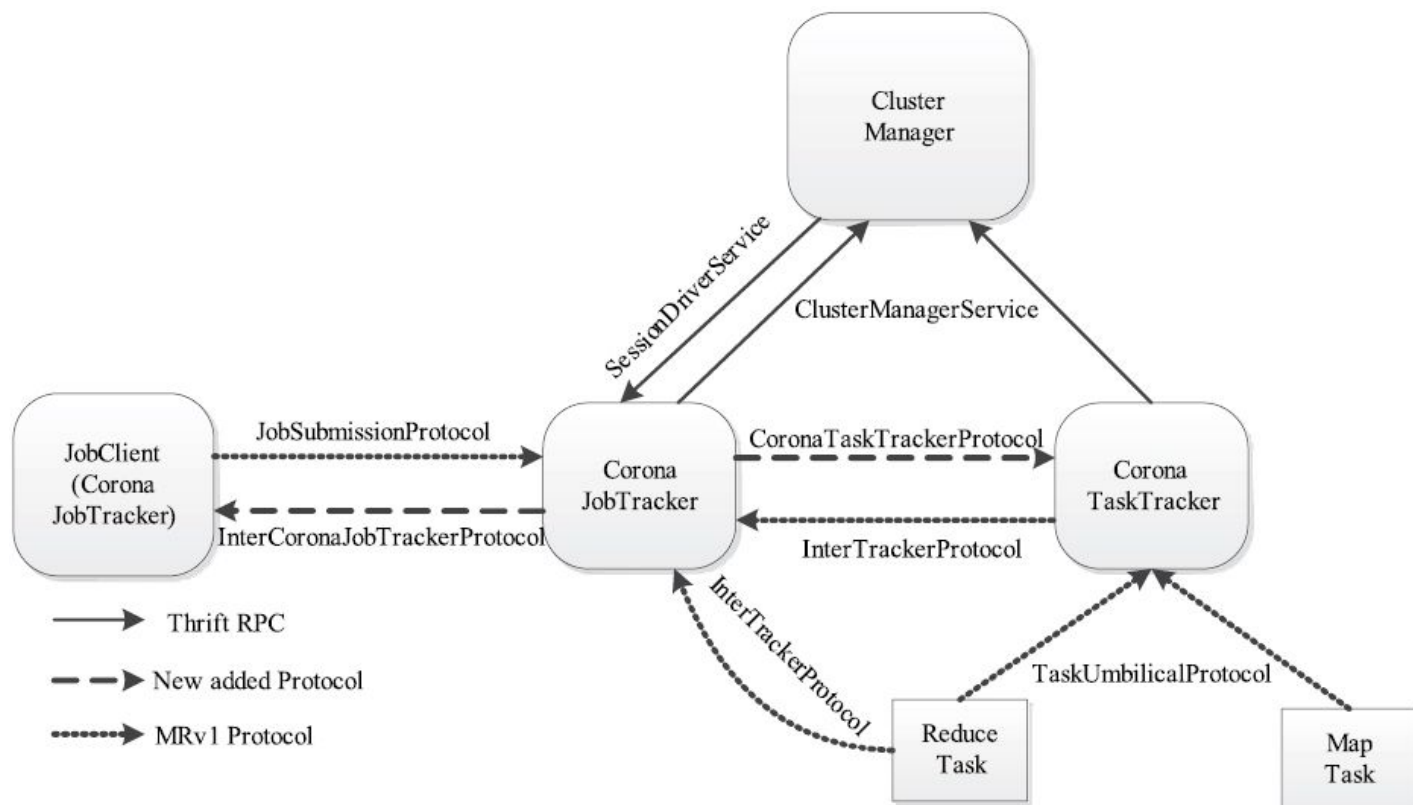


图 12-11 Corona中的RPC协议

[1] 源代码见：<https://github.com/facebook/hadoop-20/tree/master/src/contrib/corona>

[2] Under the Hood: Scheduling MapReduce jobs more efficiently with Corona

[3] 在Corona中，Reduce Task可以直接与CoronaJobTracker通信，以获取已经运行完成的Map Task列表，而不必通过CoronaTaskTracker间

接获取。

本小节介绍Hadoop Corona的工作流程。与YARN类似，当用户提交一个作业后，Hadoop Corona分两个阶段运行该作业。

第一个阶段是作业提交，实际上就是启动CoronaJobTracker的过程。根据作业大小不同，Corona JobTracker采用不同的启动模式。如果一个作业的Map Task数目小于一定阈值，CoronaJobTracker将直接启动在客户端中，这被称为本地启动模式，如图12-12所示。很显然，该模式可大大降低小作业运行延迟。另外一种远程启动模式。在该模式下，客户端首先为CoronaJobTracker（向ClusterManager）申请资源，然后在某个CoronaTaskTracker上启动CoronaJobTracker。这种模式将带来一定的运行延迟，本小节将重点介绍该启动模式。

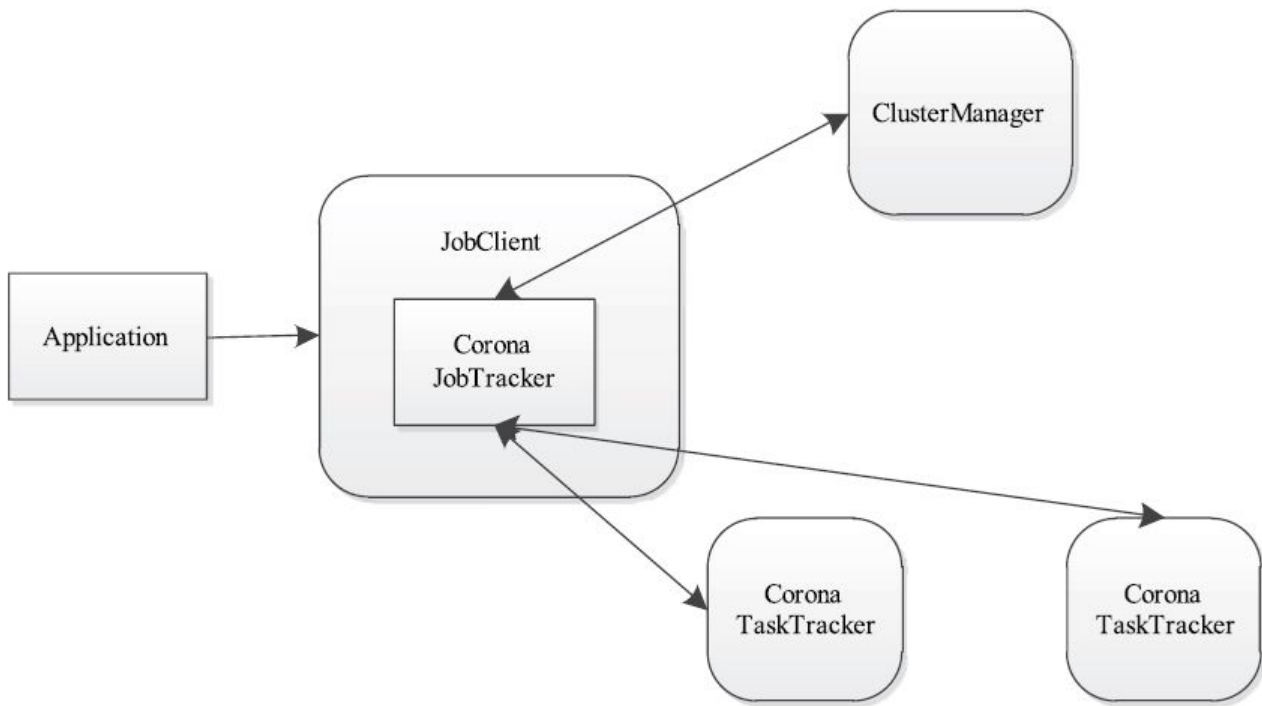


图 12-12 Corona中的本地启动模式

第二个阶段是资源申请与任务启动。在该阶段中，CoronaJobTracker不断向ClusterManager申请资源，要求CoronaTaskTracker启动任务，并监控任务的运行过程，直到所有任务运行完成。

接下来，我们重点介绍大作业（Map Task数目大于一定阈值）的提交和运行过程。

1.作业提交

作业提交过程实际上就是启动CoronaJobTracker的过程。为了与MRv 1兼容，Hadoop Corona仍由JobClient提交作业，但里面的代码已修改过。JobClient可选择将作业提交到MRv 1的JobTracker上还是Corona中。如果提交到JobTracker上，则作业运行在MRv 1中；否则，JobClient内部会创建一个CoronaJobTracker对象，然后由CoronaJobTracker（运行在转发模式下）负责提交作业。之后过程如图12-13所示，大致可分别以下几个步骤：

- 步骤1 CoronaJobTracker内部创建代理对象RemoteJITProxy，由它与ClusterManager和CoronaTaskTracker通信，为CoronaJobTracker（运行在远程模式下）申请资源并启动它。
- 步骤2 RemoteJITProxy收到启动CoronaJobTracker的请求后，首先需向ClusterManager申请资源。
- 步骤3 ClusterManager中的资源调度器为CoronaJobTracker分配对应的资源量，并返回给RemoteJITProxy。
- 步骤4 RemoteJITProxy根据收到的资源描述信息（包括资源所在节点，资源量等信息），与对应的CoronaTaskTracker通信，要求它启动CoronaJobTracker。
- 步骤5 CoronaTaskTracker成功启动CoronaJobTracker后，告诉RemoteJITProxy，然后再由RemoteJITProxy告诉客户端。

步骤6 客户端得知CoronaJobTracker启动成功后，向RemoteJTProxy提交作业，然后由RemoteJTProxy进一步将作业提交到刚刚启动的CoronaJobTracker上。

至此，一个作业正式提交成功。

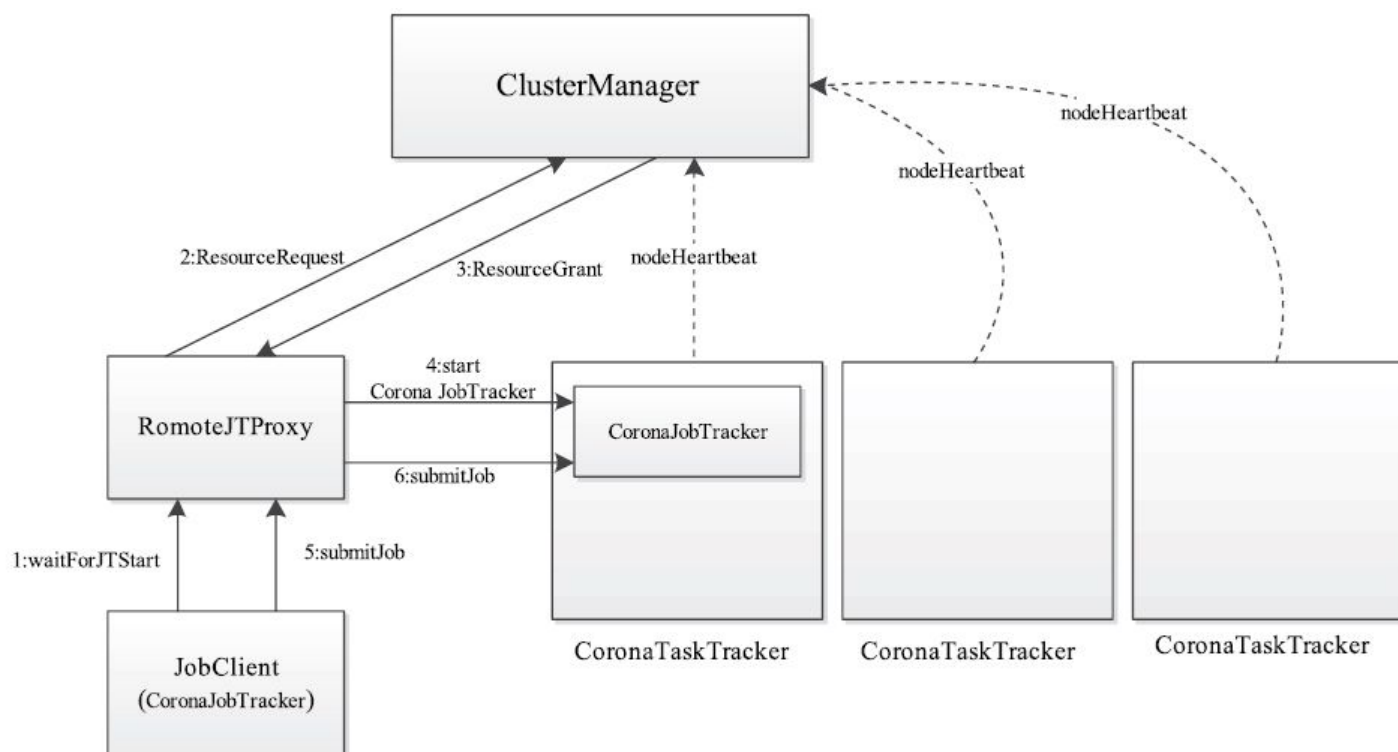


图 12-13 CoronaJobTracker启动过程

需要注意的是，整个过程涉及两个CoronaJobTracker：第一个运行在转发模式下，负责提交作业和转发客户端的各种请求；第二个运行在远程模式下，负责申请资源和监控作业运行状态。

2.资源申请与任务启动

CoronaJobTracker负责为作业申请资源，并与CoronaTaskTracker通信，要求它运行Task。总之，CoronaJobTracker功能如下。

❑ 向ClusterManager申请资源。

❑ 释放资源与资源重用：ClusterManager中的调度器支持资源抢占，可随时命令某个CoronaJobTracker释放资源，另外，CoronaJobTracker可根据需要，自行决定资源是不是重用，即某个任务运行完后，可不归还给ClusterManager，而是继续分配给其他任务。

❑ 与CoronaTaskTracker通信，以启动新任务。

❑ 任务推测执行，可参考6.6节中的具体介绍。

❑ 任务容错：当任务执行失败后，向ClusterManager重新申请资源，以重新运行该任务。

资源申请与任务启动过程如图12-14所示，主要步骤如下：

步骤1 CoronaJobTracker向ClusterManager发送资源请求。

步骤2 当某个CoronaTaskTracker出现空闲资源后，ClusterManager根据调度策略决定将该资源分配给哪些作业，并将资源描述发送给对应的CoronaJobTracker。

步骤3 CoronaJobTracker收到新分配的资源后，与对应的CoronaTaskTracker通信，要求它启动任务。

CoronaJobTracker会重复以上三个步骤，直到所有任务运行完成。此时，CoronaJobTracker通知ClusterManager释放所占用的资源，并退出。至此，一个作业运行完成。

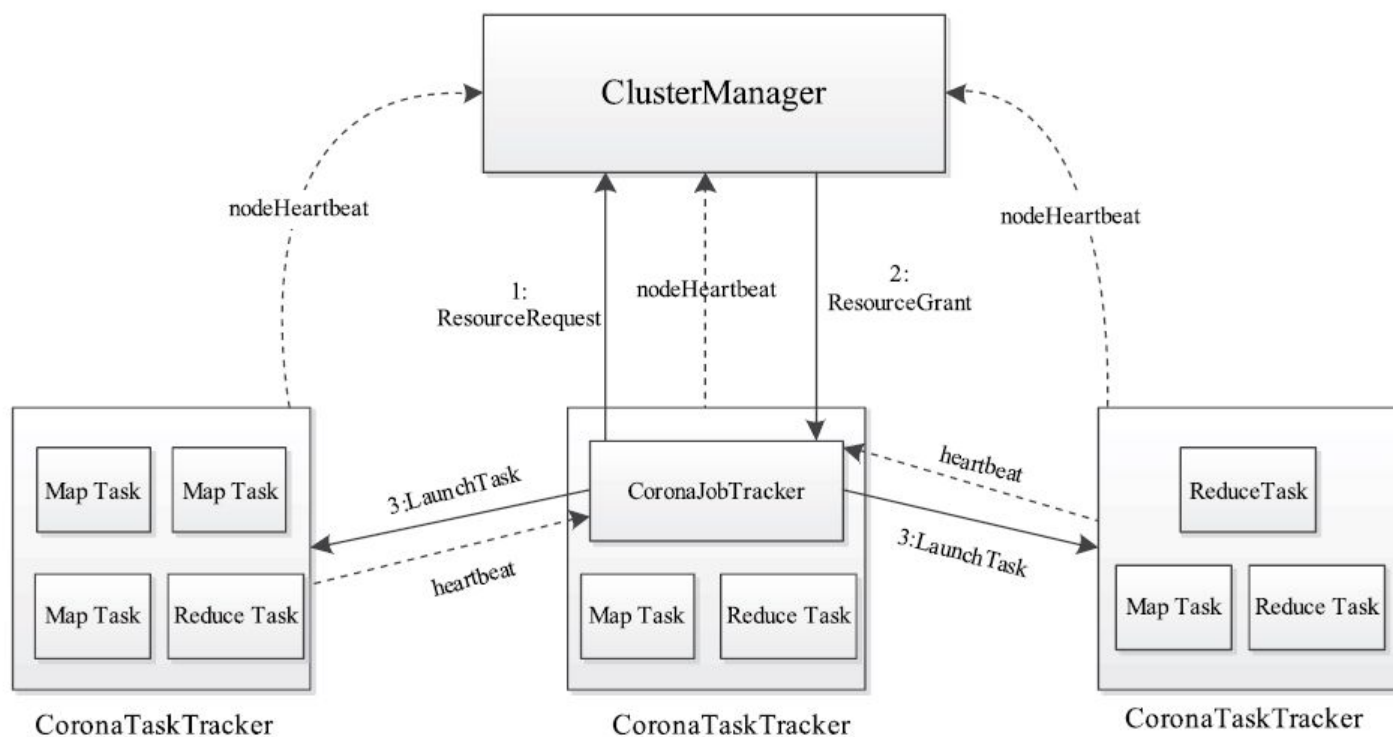


图 12-14 CoronaJobTracker资源申请与任务启动过程

12.4.3 YARN与Corona对比

尽管YARN和Corona在设计目标、设计思想等方面是一致的，但在具体实现策略上仍存在很多不同之处。表12-1对比了YARN和Corona的异同。

表 12-1 YARN 与 Corona 异同点比较

	YARN	Corona
设计目标	均是为了克服 MRv 1 在扩展性、可靠性、资源利用率等方面存在的不足	
设计思想	均是将 JobTracker 中作业控制和资源管理两个功能分开，由不同的进程完成	
资源调度模型	均是基于真实资源需求量的调度模型，而非 MRv 1 中基于 slot 的调度模型	
降低作业延迟机制	让小作业的所有任务直接在 AM Container 中运行	让小作业的 CJT 直接运行在客户端，但任务的运行还需 CJT 与 CM 和 CTT 交互
通信模型	拉式（pull-based）通信模型，AM 周期性向 RM 发送心跳询问是否分配到了新的资源	推式（push-based）通信模型，CM 将新分配的资源直接推送给 CJT，而 CJT 会进一步将新任务推送给 CTT
RPC 框架	Hadoop 自带的 RPC 框架	混合使用 Thrift RPC 和 Hadoop 自带的 RPC 框架
序列化框架	Protocol Buffers	Thrift
在线升级	采用 Zookeeper 实现	人工触发 ClusterManager 暂停服务，升级完毕后再恢复之前的状态

(续)

	YARN	Corona
可靠性	RM 可靠性由 Zookeeper 保障，AM 出现故障后由 RM 重新调度运行	截至本书出版时，CM 尚无可靠性保障，CJT 出现故障后由 CM 重新调度运行
调度器是否可插拔	可以，默认采用 FIFO 调度器 [⊖]	不可以，与 Fair Scheduler 耦合在一起

[1] YARN中的FIFO调度器存在Bug，在2.0.2-alpha和0.23.4版本中，已将默认调度器暂时切换为Capacity Scheduler，具体参考：<https://issues.apache.org/jira/browse/YARN-137>。

12.5 Apache Mesos

Mesos^[1]是诞生于UC Berkeley的一个研究项目。它的设计动机是解决编程模型和计算框架在多样化环境下，不同框架间的资源隔离和共享问题。尽管它的直接设计动机与MRv 1无关，但它的架构和实现策略与YARN或者Corona类似，因此，本节将对其进行介绍。当前有一些公司正在使用Mesos管理集群资源，比如国外的Twitter、国内的豆瓣^[2]等。

12.5.1 Apache Mesos基本框架

如图12-15所示，Apache Mesos由四个组件组成。接下来将详细对这些组件进行介绍。

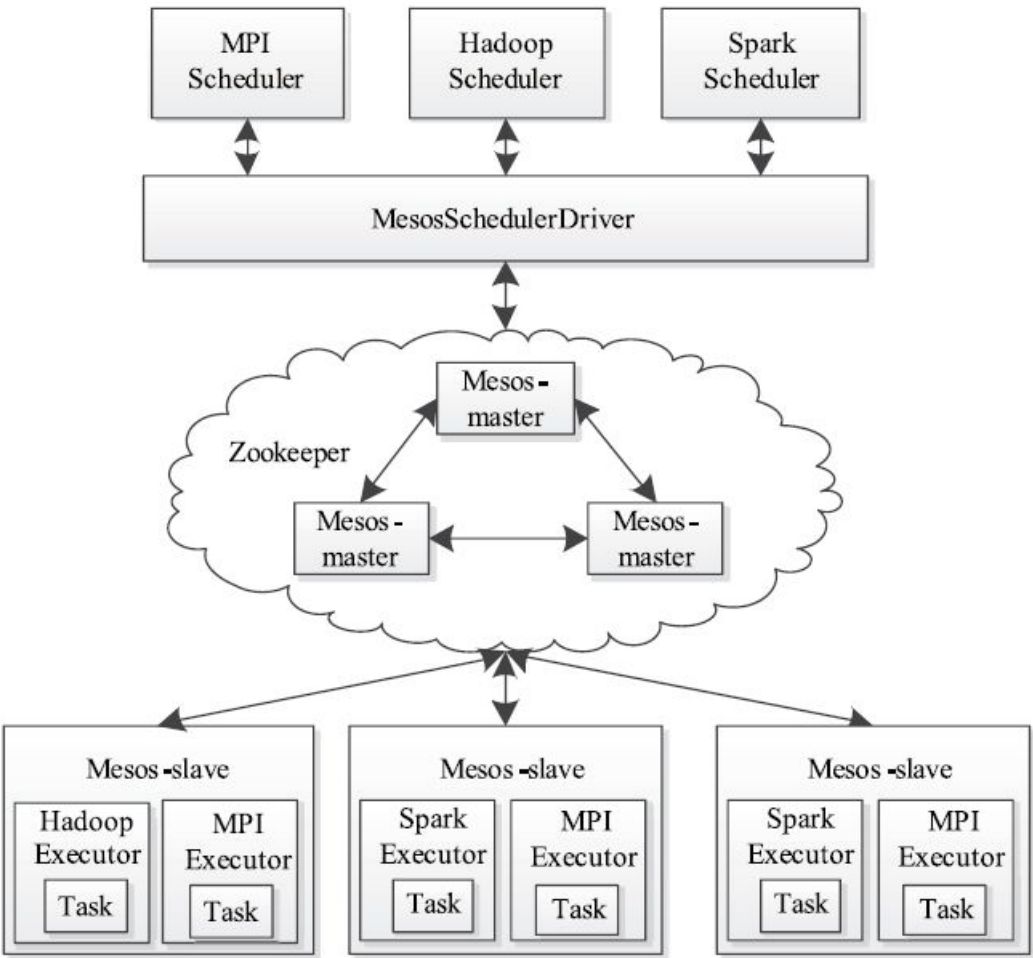


图 12-15 Mesos的基本架构

(1) Mesos-master

Mesos-master是整个系统的核心，负责管理整个系统中所有资源和接入Mesos的各种计算框架（framework），并将Mesos-slave上的资源按照某种策略分配给框架。为了防止Mesos-master出现故障后导致集群不可用，Mesos允许用户配置多个Mesos-master，并通过Zookeeper进行管理。当主Mesos-master出现故障后，Zookeeper可马上为之选择一个新的主Mesos-master。

(2) Mesos-slave

Mesos-slave负责接收并执行来自Mesos-master的命令，并定时将任务执行状态汇报给Mesos-master。Mesos-slave将节点上资源的使用情况发送给Mesos-master，由Mesos-master中的Allocator模块决定将资源分配给哪个Framework。需要注意的是，当前Mesos仅考虑了CPU和内存两种资源。当用户提交作业时，需要指定每个任务需要的CPU个数和内存量，而当任务运行时，Mesos-slave会将任务运行在包含固定资源的Linux Container中，以达到资源隔离的效果。

(3) Framework-scheduler

Framework是指外部的计算框架，如MPI、MapReduce、Spark等。这些计算框架可通过注册的方式接入Mesos，以便Mesos进行统一管理和资源分配。Mesos要求接入的框架必须有一个调度器模块Framework-scheduler，该调度器负责框架内部的任务调度。一个Framework在Mesos上的工作流程为：首先通过自己的调度器向Mesos注册，并获取Mesos分配给自己的资源，然后由自己的调度器将这些资源分配给框架中的任务。也就是说，整个Mesos系统采用了双层调度框架：第一层，由Mesos将资源分配给框架；第二层，框架自己的调度器将资源分配给内部的各个任务。当前Mesos支持三种语言编写的调度器，分别是C++、Java和Python。为了向各种调度器提供统一的接入方式，Mesos内部采用C++实现了一个MesosSchedulerDriver（调度器驱动器）。Framework的调度器可调用该driver中的接口与Mesos-master交互，完成一系列功能（如注册、资源分配等）。

(4) Framework-Executor

Framework-Executor主要用于启动框架内部的任务。由于不同的框架，启动任务的接口或者方式不同，当一个新的框架要接入Mesos时，需要编写一个对应的Executor，告诉Mesos如何启动该框架中的任务。为了给各种框架提供统一的执行器编写方式，Mesos内部采用C++实现了一个MesosExecutorDriver（执行器驱动器）。Framework可通过该驱动器的相关接口告诉Mesos启动任务的方法。

[1] Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. B.Hindman, A.Konwinski, M.Zaharia, A.Ghods, A.D.Joseph, R.Katz, S.Shenker and I.Stoica, NSDI 2011, March 2011

[2] <https://github.com/douban/dpark/>

12.5.2 Apache Mesos资源分配

Mesos中最核心的问题是如何构建一个兼具良好扩展性和性能的调度模型以支持各种计算框架。由于不同框架可能有不同的调度需求（这往往跟它的编程模型、通信模型、任务依赖关系和数据放置策略等因素相关），因此，为Mesos设计一个好的调度模型是一项极具挑战性的工作。

一种可能的解决方案是构建一个具有丰富表达能力的中央调度器。该调度器接收来自不同框架的详细需求描述，比如资源需求、任务调度顺序和组织关系等，然后为这些任务构建一个全局的调度序列。但是，在真实系统中，由于每种计算框架具有不同的调度需求，且有些框架的调度需求非常复杂，因此，提供一个具有丰富表达能力的API以捕获所有框架的需求是不太可能的，也就是说，该方案过于理想化，在真实系统中很难实现。

Mesos提供了一种简化的方案：将资源调度的控制授权给各个框架。Mesos负责按照一些简单的策略（比如FIFO, Fair等）将资源分配给各个框架，而框架内部调度器则根据一些个性化的需求将分配到的资源进一步分配给各个作业。考虑到Mesos缺少对各个框架的实际资源需求的了解，为保证框架能高效地获取到自己需要的资源，它提供了三个机制：

（1）资源拒绝

如果Mesos为某个框架分配的资源不符合它的要求，则框架可以拒绝接受该资源，直到出现满足自己需求的资源。该机制使得框架在复杂的资源约束条件下，还能够保证Mesos设计简单和具有良好的扩展性。

（2）资源过滤

每次发生资源调度时，Mesos-master均需要与Framework-scheduler进行通信，如果有些框架总是拒绝某些节点上的资源，那么额外的通信开销会使得调度性能变得低效。为避免不必要的通信，Mesos提供了资源过滤机制，允许框架只接收来自“剩余资源量大于L的Mesos-slave”或者“位于特定列表中的Mesos-slave”上的资源。

（3）资源回收

如果某个框架在一定的时间内没有为分配的资源返回对应的任务，则Mesos将回收为其分配的资源，并将这些资源重新分配给其他框架。

与YARN和Corona类似，Mesos也采用了基于真实资源需求量的调度模型。为了支持多维资源调度，Mesos采用了主资源公平调度算法（Dominant Resource Fairness, DRF）^[1]。该算法扩展了最大最小公平（Max-Min Fairness）算法^[2]，使其能够支持多维资源的调度。由于DRF被证明非常适合应用于多资源和复杂需求的环境中，因此被越来越多的系统采用，包括Apache YARN^[3]。

在DRF算法中，将所需份额（资源比例）最大的资源称为主资源，而DRF的基本设计思想则是将最大最小公平算法应用于主资源上，进而将多维资源调度问题转化为单资源调度问题，即DRF总是最大化所有主资源中最小的。其算法伪代码如下：

```
function void DRFScheduler()
R ← <r1, ..., rm>; //m种资源对应的容量
C ← <c1, ..., cm>; //已用掉的资源，初始值为0
si (i=1..n); //用户（或者框架）i的主资源所需份额，初始化为0
Ui ← <ui1, 1, ..., uim> (i=1..n) //分配给用户i的资源，初始化为0
挑选出主资源所需份额si最小的用户i;
Di ← {用户i的下一个任务需要的资源量};
if C+Di ≤ R then
//将资源分配给用户i
C ← C+Di; //更新C
Ui ← Ui + Di; //更新U
si = maxj=1..m {uij / rj};
else
return; //资源全部用完
end if
```

【实例】假设系统中共有9 CPUs和18 GB RAM，有两个用户（或者框架）分别运行了两种任务，需要的资源量分别为<1 CPU, 4 GB>和<3 CPUs, 1 GB>。对于用户A，每个任务要消耗总CPU的1/9（份额）和总内存的2/9，因而A的主资源为内存；对于用户B，每个任务要消耗总CPU的1/3和总内存的1/18，因而B的主资源为CPU。DRF将最大化所有用户的主资源，具体分配过程如表12-2所示。最终，A获取的资源量为<3 CPUs, 12 GB>，可运行3个任务；而B获取的资源量为<6 CPUs, 2 GB>，可运行2个任务。

表 12-2 DRF 算法的调度序列

调度序列	User A		User B		CPU 使用量	RAM 使用量
	资源份额	主资源份额	资源份额	主资源份额		
User B	<0,0>	0	<3/9,1/18>	1/3	3/9	1/18
User A	<1/9,4/18>	2/9	<3/9,1/18>	1/3	4/9	5/18
User A	<2/9,8/18>	4/9	<3/9,1/18>	1/3	5/9	9/18
User B	<2/9,8/18>	4/9	<6/9,2/18>	2/3	8/9	10/18
User A	<3/9,12/18>	2/3	<6/9,2/18>	2/3	1	14/18

[1] Dominant Resource Fairness: Fair Allocation of Multiple Resources Types. A.Ghods, M.Zaharia, B.Hindman, A.Konwinski, S.Shenker, and I.Stoica, NSDI 2011, March 2011

[2] Max-Min Fairness (Wikipedia) : http://en.wikipedia.org/wiki/Max-min_fairness

[3] <https://issues.apache.org/jira/browse/YARN-2>

12.5.3 MapReduce与Mesos结合

由于Hadoop MapReduce的JobTracker和TaskTracker与Mesos模型中的Framework-scheduler和Executor在设计上是完全对应的，因此，只需修改少量代码便可让MapReduce运行于Mesos之上。

为了让Mesos支持MapReduce，需为其编写Scheduler和Executor两个组件。它们的作用如下：

(1) Scheduler

Mesos利用了Hadoop调度器可插拔的特性，重新实现了一个可接入Mesos的调度器MeosScheduler，以实现框架注册、资源分配等功能。

(2) Executor

Mesos为Hadoop实现了一个Executor——FrameworkExecutor。通过该Executor, Mesos可控制TaskTracker的启动或停止。比如，如果一定时间内没有任务运行，Mesos会关闭某些节点上的TaskTracker。

为了使Hadoop能运行于Mesos之上，JobTracker启动时通过调度器向Mesos注册，这之后，资源分配过程如图12-16所示，大致分为以下几个步骤：

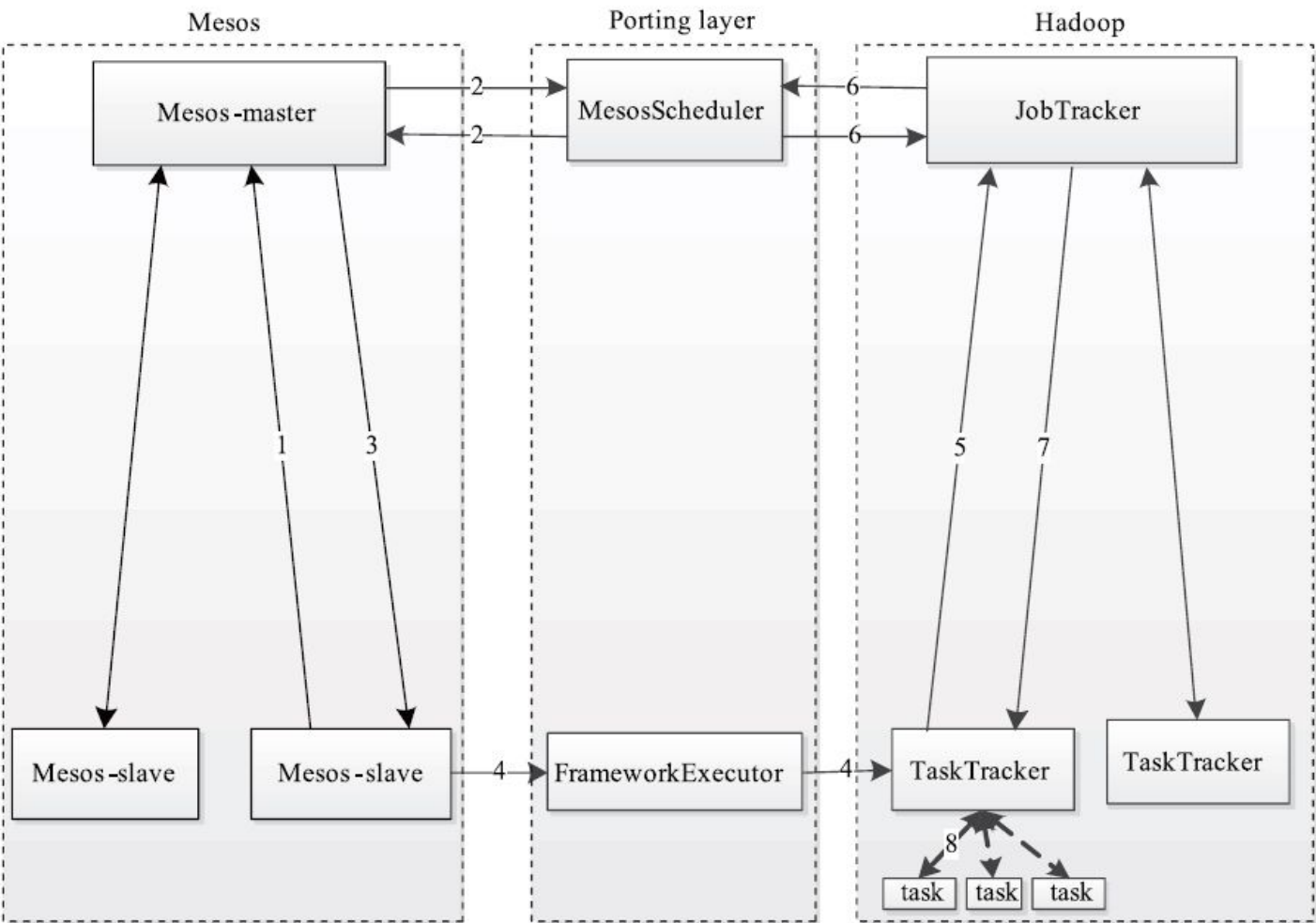


图 12-16 Hadoop在Mesos上工作流程

- 步骤1 Mesos-slave向Mesos-master汇报自己的资源使用情况，当前支持的资源类型包括CPU和内存两种。
- 步骤2 如果Mesos-slave上有空闲资源，Mesos-master会按照一定的策略将资源分配给当前向其注册的框架。如果分配给

Hadoop，则告诉其调度器，它会按照一定的策略选出若干Hadoop任务并暂时缓存起来（稍后对应的TaskTracker通过心跳领取这些任务，见步骤5），并返回一系列Mesos-task列表（该任务的作用仅是在Hadoop任务正式启动之前为它准备资源，并不用于真正的计算），其中每个Mesos-task会对应一个Hadoop任务。

步骤3 Mesos-master将Mesos-task列表发送给对应的Mesos-slave。

步骤4 Mesos-slave收到一个Mesos-task后，检查它是否是收到的第一个来自Hadoop框架的任务，如果是，则通过Executor为其启动TaskTracker。

步骤5 与正常的Hadoop集群一样，TaskTracker向JobTracker汇报心跳，以领取新的计算任务（即步骤2中缓存起来的任务）。

步骤6 JobTracker收到来自TaskTracker的心跳后，通过调度器为其分配任务。需要注意的是，此时调度器分配多少任务并不取决于该TaskTracker上有多少空闲的slot，而是取决于Mesos为其分配的资源量（在步骤2中分配的资源量）。

步骤7 JobTracker将新分配的任务返回给对应的TaskTracker。

步骤8 TaskTracker收到新任务后，启动这些任务。需要注意的是，这些任务执行过程中的状态更新不仅会告诉TaskTracker，也会通过Executor汇报给Mesos。

12.6 小结

本章介绍了下一代MapReduce的基本设计思想以及常见的三个实现：YARN、Corona和Mesos。

YARN是Apache的下一代MapReduce框架。它的基本设计思想是将JobTracker拆分成两个独立的服务：一个全局的资源管理器ResourceManager和每个应用程序特有的ApplicationMaster。其中，ResourceManager负责整个系统的资源管理和分配，而ApplicationMaster则负责单个应用程序的管理。

与YARN一样，Corona也是将JobTracker拆分成了两个独立的服务：一个全局的资源管理器ClusterManager和每个应用程序特有的CoronaJobTracker。其中，ClusterManager负责整个系统的资源管理和分配，而CoronaJobTracker则负责单个应用程序的管理。

Mesos诞生于UC Berkeley的一个研究项目。它的设计动机是解决编程模型和计算框架在多样化的环境下的框架间资源隔离和共享问题。尽管它不是直接从MRv 1衍化而来的，但它具备了下一代MapReduce的基本特征。

尽管YARN、Corona和Mesos诞生于不同的公司或者研究机构，但它们的架构却大同小异，表12-3给出了它们内部基本组件的对应关系。

表 12-3 YARN、Corona 和 Mesos 基本组件对应关系

YARN	Corona	Mesos	功 能
Resource Manager	Cluster Manager	Mesos Master	负责整个集群的资源管理和调度。
Node Manager	Corona TaskTracker	Mesos Slave	负责单个节点的资源管理（资源隔离、资源使用汇报等）、任务管理（启动、杀死等）和任务运行进度汇报等。
		Framework- executor	
Application Master	Corona JobTracker	Framework-scheduler	负责单个应用程序的管理和资源的二次调度（将资源分配给内部的各个任务）。

随着应用需求的多样化和数据量的增加，下一代MapReduce必将逐渐替代MRv 1，被越来越多的公司采用。

附录A 安装Hadoop过程中可能存在的问题及解决方案

问题1 在Windows中运行Hadoop时，出现以下异常：

```
ERROR security.UserGroupInformation (UserGroupInformation.java: doAs (1086)) -PrivilegedActionException as:
dong cause: java.io.IOException: Failed to set permissions of path: D:\hadoop\hadoop-
1.0.0\build\test\mapred\staging\trss1723775845\.staging to 0700
```

【产生原因】

由于Hadoop 1.0.0中独特的修改目录权限^[1]的方法，使得编译后的jar包不能直接运行于Windows环境。

【解决方法】

最简捷的方法是，对src\core\org\apache\hadoop\fs\FileUtil.java代码做以下修改（注释三行代码）：

```
private static void checkReturnValue (boolean rv, File p,
FsPermission permission
) throws IOException{
if (! rv) {
//throw new IOException ("Failed to set permissions of path: "+p+
//
"to"+
//
String.format ("%04o", permission.toShort ()) );
}}
```

修改之后重新编译Hadoop源代码。

问题2 Hadoop eclipse plugin报错：“Error: failure to login”

【产生原因】

直接编译代码后生成的jar包由于缺少一些依赖的lib文件，不能直接使用。

【解决方法】

对配置文件进行部分修改，修改的文件涉及src/contrib/eclipse-plugin目录下的build.xml和META-INF/MANIFEST.MF文件。

修改build.xml如下（在文件末尾添加以下内容）：

```
<copy file="${hadoop.root}/lib/commons-configuration-1.6.jar"todir="${build.dir}/lib"verbose="true"/>
<copy file="${hadoop.root}/lib/commons-lang-2.4.jar"todir="${build.dir}/lib"verbose="true"/>
<copy file="${hadoop.root}/lib/jackson-core-asl-1.0.1.jar"todir="${build.dir}/lib"verbose="true"/>
<copy file="${hadoop.root}/lib/jackson-mapper-asl-1.0.1.jar"todir="${build.dir}/lib"verbose="true"/>
<copy file="${hadoop.root}/lib/commons-httpclient-3.0.1.jar"todir="${build.dir}/lib"verbose="true"/>
```

将META-INF/MANIFEST.MF文件中的Bundle-ClassPath属性修改为：

```
Bundle-ClassPath: classes/, lib/hadoop-core.jar, lib/commons-configuration-1.6.jar, lib/commons-lang-2.4.jar,
lib/jackson-core-asl-1.0.2.jar, lib/jackson-mapper-asl-1.0.2.jar, lib/commons-httpclient-3.0.1.jar
```

经过上面编译之后，产生的jar包位于build/contrib/eclipse-plugin目录下，将该jar包拷贝到Eclipse的plugins目录下，重启Eclipse即可。

问题3 Windows环境下使用Cygwin终端启动Hadoop时，出现错误“ssh: connect to host localhost port 22: Connection refused”

【产生原因】

Windows系统下，ssh命令需要管理员权限才能使用。

【解决方案】

方案一 永久修改ssh启动权限

步骤1: Windows环境下，依次选择开始→运行→输入services.msc。

步骤2: 右击CYGWIN sshd，并依次选择属性→登录→“此账户”→浏览→高级→立即查找→选择账户名（必须为管理员权限）→输入密码→确定。

步骤3: 重启CYGWIN sshd服务。

方案二 以管理员身份打开Cygwin终端

右击Cygwin可执行文件，单击“以管理员身份运行”。

[1] <https://issues.apache.org/jira/browse/MAPREDUCE-3555>

附录B Hadoop默认HTTP端口号以及HTTP地址

为了便于用户通过这些Web UI查看Hadoop当前运行状态，Hadoop对外提供了一些访问URL。这些URL的默认端口号汇总如表B-1所示。

表 B-1 默认端口号汇总

Hadoop 模块	守护进程	默认端口号	配置参数
HDFS	NameNode	50070	dfs.http.address
	SecondNameNode	50090	dfs.secondary.http.address
	DataNodes	50075	dfs.datanode.http.address
MapReduce	JobTracker	50030	mapred.job.tracker.http.address
	TaskTracker	50060	mapred.task.tracker.http.address

Hadoop守护进程向用户提供的HTTP地址有以下几项。

□/logLevel: 查看和修改Hadoop源代码类的输出日志级别。

□/stacks: 显示当前所有线程的栈轨迹，主要用于调试。

□/metrics: 显示Hadoop中所有Metrics信息。

□/logs: 显示日志目录中的所有日志文件，可供用户查看日志或者下载日志使用。举例说明：

□显示JobTracker（ip为10.10.10.100）所在机器上Hadoop产生的日志：

http://10.10.10.100: 50030/logs

□显示TaskTracker（ip为10.10.10.111）上的Metrics信息：

http://10.10.10.111: 50060/metrics

参考资料

【参考书籍】

- [1]Tom White. Hadoop权威指南[M].2版.周敏奇, 王晓玲, 金澈清, 钱卫宁, 译.北京: 清华大学出版社, 2011.
- [2]Chuck Lam. Hadoop实战[M].韩冀中.北京: 人民邮电出版社, 2011.
- [3]Eric Sammer. Hadoop Operations.O'Reilly Media, 2012.
- [4]孙玉琴.Java网络编程精解[M].北京: 电子工业出版社, 2007.
- [5]Ron Hitchens. Java NIO.O'Reilly Media, 2002.
- [6]George Coulouris, Jean Dollimore, Tim Kindberg.分布式系统概念与设计[M].金蓓弘, 等译.北京: 机械工业出版社, 2004.
- [7]Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.设计模式: 可复用面向对象软件的基础[M].李英军, 等译.北京: 机械工业出版社, 2000.
- [8]Eric Freeman, Elisabeth Freeman, Kathy Sterra, Bert Bates. O'Reilly公司.Head First设计模式[M].北京: 中国电力出版社, 2007.

【参考论文】

- [1]J. Dean and S.Ghemawat, "Mapreduce: simplified data processing on large clusters, "in Proceedings of the 6th conference on Symposium on Operating Systems Design& Implementation-Volume 6.Berkeley, CA, USA: USENIX Association, 2004, pp.107-113.
- [2]Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system.In 19th Symposium on Operating Systems Principles, pages 29-43, Lake George, New York, 2003.
- [3]Jorge-Arnulfo Quiané-Ruiz, Christoph Pinkel, Jörg Schad, Jens Dittrich.RAFTing MapReduce: Fast recovery on the RAFT.In Serge Abiteboul, Klemens Böhm, Christoph Koch, Kian-Lee Tan, editors, Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany.
- [4]Matei Zaharia, Andrew Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica, Improving MapReduce Performance in Heterogeneous Environments, 8th USENIX Symposium on Operating Systems Design Implementation, pp.29-42, San Diego, CA, December, 2008.
- [5]Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, Song Guo, "SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment, "Computer and Information Technology (CIT), 2010 IEEE 10th International Conference.
- [6]梁李印, "阿里Hadoop集群架构及服务体系", PPT, Hadoop与大数据技术大会 (HBTC 2012) .
- [7]A. Ghodsi, M.Zaharia, B.Hindman, A.Konwinski, S.Shenker, and I.Stoica.Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.In USENIX NSDI, 2011.
- [8]Hong Mao, Shengqiu Hu, Zhenzhong Zhang, Limin Xiao, Li Ruan: A Load-Driven Task Scheduler with Adaptive DSC for MapReduce. GreenCom 2011: 28-33.
- [9]Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, Dhiraj Sehgal. Hadoop Acceleration through Network Levitated Merging.SC11.Seattle, WA.

- [10]Herodotos Herodotou. Hadoop Performance Models, Technical Report, CS-2011-05, Computer Science Department Duke University.
- [11]连林江: “百度分布式计算技术发展”, 2012.07.08.
- [12]M. Zaharia, D.Borthakur, J.S.Sarma, K.Elmeleegy, S.Shenker, and I.Stoica, “Job scheduling for multi-user mapreduce clusters, ”EECS Department, University of California, Berkeley, Tech.Rep., Apr 2009.
- [13]M. Zaharia, D.Borthakur, J.S.Sarma, K.Elmeleegy, S.Shenker, and I.Stoica, “Efficient Fair Scheduling for MapReduce”, PPT.
- [14]Todd Lipcon, Cloudera, “Optimiziong MapReduce Job Performance”, Hadoop Summit 2012.
- [15]M. Zaharia, D.Borthakur, J.Sen Sarma, K.Elmeleegy, S.Shenker, and I.Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling”in Proc.of EuroSys.ACM, 2010, pp.265-278.
- [16]Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop.In JSSPP'10: 15th Workshop on Job Scheduling Strategies for Parallel Processing, 2010.
- [17]J. Polo, D.Carrera, Y.Becerra, J.Torres, E.Ayguade and, M.Steinder, and I.Whalley, “Performance-driven task co-scheduling for mapreduce environments, ”in Network Operations and Management Symposium (NOMS) , 2010 IEEE, 2010, pp.373-380.
- [18]Faraz Ahmad, Seyong Lee, Mithuna Thottethodi and T. N.Vijaykumar, “MapReduce with Communication Overlap (MaRCO) ”, ECE Technical Reports, 2007.11.01.
- [19]Owen O'Malley, “Plugging the Holes: Security and Compatibility”, PPT.
- [20]Kerberos认证协议的教学设计, 计算机系统与网络安全设计课题组, 电子科技大学科学与工程学院.
- [21]Owen O'Malley, Kan Zhang, Sanjay Radia, Ram Marti, and Christopher Harrell, “Hadoop Security Design”, Yahoo!
- [22]Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. B.Hindman, A.Konwinski, M.Zaharia, A.Ghodsi, A.D.Joseph, R.Katz, S.Shenker and I.Stoica, NSDI 2011, March 2011.
- [23]Dominant Resource Fairness: Fair Allocation of Multiple Resources Types. A.Ghodsi, M.Zaharia, B.Hindman, A.Konwinski, S.Shenker, and I.Stoica, NSDI 2011, March 2011.
- [24]“yarn (hadoop2) 框架的一些软件设计模式”, CSDN.
- [25]AMD white paper: “Hadoop Performance Tuning Guide”.
- 【参考Hadoop Jira [\[1\]](#)】
- [1]HDFS-1052: HDFS scalability with multiple namenodes.
- [2]HDFS-1623: High Availability Framework for HDFS NN. HDFS-200: In HDFS, sync()not yet guarantees data available to the new readers.
- [3]HDFS-265: Revisit append.
- [4]HDFS-503: Implement erasure coding as a layer on HDFS.

- [5]HDFS-245: Create symbolic links in HDFS.
- [6]HADOOP-4487: Security features for Hadoop.
- [7]HADOOP-6332: Large-scale Automated Test Framework.
- [8]HADOOP-1230: Replace parameters with context objects in Mapper, Reducer, Partitioner, InputFormat, and OutputFormat classes.
- [9]MAPREDUCE-334: Change mapred. lib code to use new api.
- [10]HADOOP-1722: Make streaming to handle non-utf8 byte array.
- [11]HADOOP-7775: RPC Layer improvements to support protocol compatibility.
- [12]HADOOP-7347: IPC Wire Compatibility.
- [13]HADOOP-4797: RPC Server can leave a lot of direct buffers.
- [14]HDFS-2676: Remove Avro RPC.
- [15]HDFS-2058: DataTransfer Protocol using protobufs.
- [16]MAPREDUCE-1099: Setup and cleanup tasks could affect job latency if they are caught running on bad nodes.
- [17]MAPREDUCE-463: The job setup and cleanup tasks should be optional.
- [18]MAPREDUCE-744: Support in DistributedCache to share cache files with other users after HADOOP-4493.
- [19]HADOOP-153: skip records that fail Task.
- [20]HADOOP-2141: speculative execution start up condition based on completion time.
- [21]MAPREDUCE-2657: TaskTracker should handle disk failures.
- [22]MAPREDUCE-1906: Lower minimum heartbeat interval for tasktracker> Jobtracker.
- [23]HADOOP-3245: Provide ability to persist running jobs (extend HADOOP-1876) .
- [24]MAPREDUCE-873: Simplify Job Recovery.
- [25]MAPREDUCE-211: Provide a node health check script and run it periodically to check the node health status.
- [26]HADOOP-4305: repeatedly blacklisted tasktrackers should get declared dead.
- [27]HADOOP-5643: Ability to blacklist tasktracker.
- [28]MAPREDUCE-2657: TaskTracker should handle disk failures.
- [29]MAPREDUCE-2415: Distribute TaskTracker userlogs onto multiple disks.
- [30]HADOOP-692: Rack-aware Replica Placement.
- [31]MAPREDUCE-2415: Distribute TaskTracker userlogs onto multiple disks.

- [32]MAPREDUCE-2364: Shouldn't hold lock on rjob while localizing resources.
- [33]HADOOP-5883: TaskMemoryMonitorThread might shoot down tasks even if their processes momentarily exceed the requested memory.
- [34]MAPREDUCE-1221: Kill tasks on a node if the free physical memory on that machine falls below a configured threshold.
- [35]MAPREDUCE-211: Provide a node health check script and run it periodically to check the node health status.
- [36]MAPREDUCE-4039: Sort Avoidance.
- [37]MAPREDUCE-4049: plugin for generic shuffle service.
- [38]HADOOP-331: map outputs should be written to a single output file with an index.
- [39]MAPREDUCE-240: Improve the shuffle phase by using the“connection: keep-alive”and doing batch transfers of files.
- [40]MAPREDUCE-2841: Task level native optimization.
- [41]MAPREDUCE-64: Map-side sort is hampered by io. sort.record.percent.
- [42]HADOOP-1965: Handle map output buffers better.
- [43]MAPREDUCE-1380: Adaptive Scheduler.
- [44]MAPREDUCE-1439: Learning Scheduler.
- [45]MAPREDUCE-4360: Capacity Scheduler Hierarchical leaf queue does not honor the max capacity of container queue.
- [46]MAPREDUCE-2905: CapBasedLoadManager incorrectly allows assignment when assignMultiple is true (was: assignmultiple per job) .
- [47]HADOOP-4487: Security features for Hadoop.
- [48]MAPREDUCE-2405: MR-279: Implement uber-AppMaster (in-cluster LocalJobRunner for MRv2) .
- [49]YARN-3: Add support for CPU isolation/monitoring of containers.
- [50]YARN-2: Enhance CS to schedule accounting for both memory and cpu cores.
- [51]YARN-137: Change the default scheduler to the CapacityScheduler.
- [52]MAPREDUCE-211: Provide a node health check script and run it periodically to check the node health status.
- [53]MAPREDUCE-1906: Lower default minimum heartbeat interval for tasktracker> Jobtracker.
- [54]MAPREDUCE-2355: Add an out of band heartbeat damper.
- [55]HADOOP-3136: Assign multiple tasks per TaskTracker heartbeat.
- [56]HADOOP-7206: Integrate Snappy compression.
- [57]HADOOP-7714: Umbrella for usage of native calls to manage OS cache and readahead.

【参考网络资源】

- [1]Apache log4j网址: <http://logging.apache.org/log4j/index.html>.
- [2]Nutch官方网站: <http://nutch.apache.org/>.
- [3]Lucene官方网站: <http://lucene.apache.org/>.
- [4]HDFS RAID介绍: <http://wiki.apache.org/hadoop/HDFS-RAID>.
- [5]An update on Apache Hadoop 1.0: <http://blog.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/>.
- [6]Fault Inject框架介绍: http://hadoop.apache.org/docs/hdfs/r0.21.0/faultinject_framework.html.
- [7]Spark官方主页: <http://www.spark-project.org/>.
- [8]Oozie官方主页: <http://incubator.apache.org/oozie/>.
- [9]排序基准: <http://sortbenchmark.org/>.
- [10]HBase官方主页: <http://hbase.apache.org/>.
- [11]Hive官方主页: <http://hive.apache.org/>.
- [12]Pig官方主页: <http://pig.apache.org/>.
- [13]Cascading官方主页: <http://www.cascading.org/>.
- [14]Azkaban官方主页: <http://sna-projects.com/azkaban/>.
- [15]Using Hadoop IPC/RPC for distributed applications: <http://www.supermind.org/blog/520>.
- [16]Architecture of a Highly Scalable NIO-Based Server: <http://today.java.net/pub/a/today/2007/02/13/architecture-of-highly-scalable-nio-server.html>.
- [17]New I/O APIs: <http://docs.oracle.com/javase/1.4.2/docs/guide/nio/>.
- [18]Thrift官方主页: <http://thrift.apache.org/>.
- [19]Protocol Buffer官方主页: <http://code.google.com/p/protobuf/>.
- [20]Avro官方主页: <http://avro.apache.org/>.
- [21]“在Hadoop上调试HadoopStreaming程序的方法详解”, 道凡.
- [22]Hanborq optimized Hadoop Distribution: <https://github.com/hanborq/hadoop>.
- [23]MapReduce: 详解Shuffle过程: <http://langyu.iteye.com/blog/992916>.
- [24]快速排序及优化: <http://rdc.taobao.com/team/jm/archives/252>.
- [25]Hadoop源代码分析: <http://caibinbupt.iteye.com/>.
- [26]nativetask代码及文档: <https://github.com/decster/nativetask>.

[27]HOD说明文档: http://hadoop.apache.org/docs/stable/hod_scheduler.html.

[28]Torque官方网站: <http://www.adaptivecomputing.com/products/open-source/torque/>.

[29]Capacity Scheduler说明文档: http://hadoop.apache.org/docs/stable/capacity_scheduler.html.

[30]Fair Scheduler说明文档: http://hadoop.apache.org/docs/stable/fair_scheduler.html.

[31]Max-Min Fairness (Wikipedia): http://en.wikipedia.org/wiki/Max-min_fairness.

[32]Kerberos Wiki介绍: <http://jianlee.ylinux.org/Computer/Wiki/kerberos.html>.

[33]Cloudera CDH3文档: <https://ccp.cloudera.com/display/CDHDOC/CDH3+Security+Guide>.

[34]YARN与Mesos比较: <http://www.quora.com/How-does-YARN-compare-to-Mesos>.

[35]Hortonworks官方博客: <http://hortonworks.com/blog/>.

[36]Cloudera官方博客: <http://blog.cloudera.com/blog/>.

[37]Facebook Hadoop代码: <https://github.com/facebook/hadoop-20>.

[38]Mesos官方网站: <http://www.mesosproject.org/>.

[39]<http://www.oberhumer.com/opensource/lzo/>.

[40]<http://code.google.com/p/snappy/>.

[41]<https://github.com/toddlipton/hadoop-lzo>.

[1] Hadoop Jira是Hadoop的项目管理系统, 通过它可追踪一些问题的解决过程。比如问题“HADOOP-7775”, 可通过网址“<https://issues.apache.org/jira/browse/HADOOP-7775>”查看。